Sorbonne Université
Master Informatique - BioInformatique et Modélisation

Développement d'un algorithme basé sur les 'string kernel' et architectures parallèles pour identifier les gènes codant les anticorps

Encadrante: Juliana BERNARDES

Auteurs
Elio EL GHOUL et Antoine KHOW

22 mai 2022

Table des matières

Chapitre 1

Introduction

Pour se défendre contre des infections sérieuses, le corps humain possède une immunité adaptative qui va identifier l'agent pathogène responsable et produire une réponse défensive adaptée. Dans le cadre de ce projet, nous nous sommes intéressés à une partie de cette immunité adaptative, les anticorps. Les anticorps sont composés de chaînes de protéines contenant une partie constante et une partie variable. La région variable des anticorps est codée par des gènes situés à divers endroits du génome, ces gènes possèdent 3 types de segments : gènes Variables (V), gènes de Diversité (D) et gènes de Jonction (J). Chaque segment possède également une certaine diversité, en effet, il existe environ 50 gènes V, 30 gènes D et 6 gènes J. L'idée de notre projet est donc d'identifier ces gènes dans le génome. Mais la tâche est complexe. En effet il y 9000 combinaisons possibles de VDJ et la jonction entre ces gènes est composée de nucléotides sujets à des délétions ou insertions augmentant encore davantage la diversité de la région variable. Il y a également très souvent de nombreux anticorps à analyser pour identifier ces gènes ce qui peut mener à une recherche très lente. L'enjeu est donc double ; il faut créer un algorithme capable d'identifier avec précision les gènes V, D et J et l'optimiser pour éviter un temps d'exécution trop long. L'algorithme développé durant le projet permet de calculer le taux de vrais positifs des trois segments des régions variables des anticorps : V, D, J. Nous avons été encadré tout le long du projet par Juliana BERNARDES et le projet peut être trouvé sur le dépôt Github suivant : https ://github.com/julibinho/VDJ-Kernel

Chapitre 2

État de l'art

A ce jour, il existe déjà un certain nombre d'algorithmes capables d'effectuer le travail que nous cherchons à coder (Figure 2.1). Certains algorithmes sont sous la forme d'un programme à télécharger, paramétrer et exécuter, d'autres sont des outils en ligne proposant les mêmes fonctionnalités et paramètres.

Methods	IGHV (%)	IGHD (%)	IGHJ (%)
DSab-origin	94.27	62.89	93.51
IgBLAST [10]	96.05	55.64	94.47
IgSCUEAL [16]	99.57	46.95	98.73
IMGT/V-Quest [11]	96.30	53.87	93.38
Vdjalign [14]	83.01	61.48	92.64
iHMMune [13]	90.90	57.70	92.51
Clonanalyst [30]	77.13	58.34	89.20
vdj [29]	75.96	57.35	89.39
SoDa [33]	91.33	54.95	82.82

FIGURE 2.1 – Table de comparaison des vrais positifs sur divers algorithmes pour des séquences à 40 mutations simulées.

Si l'on constate des performances très satisfaisantes sur les pourcentages de vrais positifs au niveau des V et des J, les pourcentages des D semblent être beaucoup plus instables. En effet, nous verrons plus tard pour quelles raisons les D semblent être plus difficiles à identifier que les autres gènes. Un autre point intéressant à noter est le temps d'exécution de ces algorithmes. Si la précision est au rendez-vous, il n'est pas garanti que le temps d'exécution soit convenable. Les deux points majeurs du projet sont donc le temps et la précision. Il faudra écrire un programme optimisé qui puisse donner des résultats rapidement et ces résultats devront être les plus précis possible. A noter qu'il est difficile de comparer le temps relatif de notre projet avec les outils en ligne simplement car ces outils sont exécutés à distance sur d'autres machines. Les délais de connexion internet, débits serveurs, puissance de la machine à distance étant des fluctuations non négligeable biaisant les performances.

Sur le long du projet, nous avons notamment travaillé avec le logiciel DSab car il s'agit d'un des algorithmes le plus récent. En effet, DSab est capable de détecter les D avec une très grande précision. A la base, nous devions incorporer DSab à l'algorithme du projet

afin de pouvoir ”matcher” efficacement les D et donc se concentrer sur les V et les J. Au final, l'algorithme avait un temps d'exécution beaucoup trop long et nous avons donc décidé de le laisser de côté et de créer notre propre algorithme à partir de zéro pour les D afin d'essayer de nous rapprocher de ses performances.

Chapitre 3

Construction de l'algorithme du projet

3.1 Identification des gènes hits

Vu que la région des anticorps est constituée des segments de gène V, D et J, la séquence query est artificiellement divisée en trois blocs : bloc V (variable V), bloc NDN (diversity D and additions), et bloc J (joining J) (Figure 3.1). Afin de séparer ces trois parties, nous avons tout d'abord identifié les gènes hits pour le V et le J en les comparant avec les gènes de références "human IGHV" et "human IGHJ" respectivement de IMGT en appliquant l'algorithme du projet. Après avoir identifié les meilleurs gènes hits, nous avons enlevé les blocs V et J de la séquence query. Le NDN block restant a été étudié avec l'algorithme du projet combiné avec la méthode des Hidden Markov Model. L'identification des gènes hits pour le D a été aussi réalisée avec les gènes de références "human IGHD" de IMGT. Tout le long du rapport, les gènes extraits des répertoires "human IGHV", "human IGHD" et "human IGHJ" sont nommés séquences de références et les gènes que nous souhaitons diviser sont nommés séquences query.

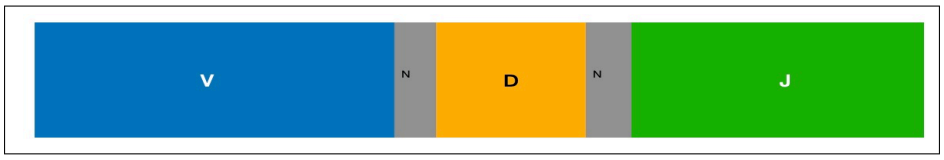

FIGURE 3.1 – Schéma simplifié des trois blocs V, D et J

3.2 Données simulées

Pour tester notre algorithme et observer ses performances, nous utilisons des données issues de simulations. Ces simulations sont, comme leur nom l'indique, des séquences simulées de région variable des anticorps. Ces séquences sont, à proprement parler, des modèles pour représenter cette région variable. Elles en ont donc les mêmes caractéristiques nous permettant ainsi de tester notre algorithme dessus. Les données sont contenues dans des fichiers .fasta ou chaque séquence est représentée sous la forme suivante :

>S1|IGHV3-30-3*01|IGHD5-12*01|IGHJ1*01|40
CAAGTGCAGCTGCTGGAGTCTAGGGGAGGCGTGGTCCAGCCTGGGCGGTCC
CTGTGGCCCTCCTCTGCAGCCTCTGGATTCACCCTCAATAGTAATTCTCTGC
ACTGGGTCCGCCAGGCTCCAGTCAAGGGGCTGGATAGGGTGGCAGTCATATC
ATATGATGGAAGAACTATCTGCTACGCAGACTCCGTGAAGGGCCGATTCACCC
TCTCCAGAGACAATTCCAAGAACACGCTGAATCCGCAAATGAACAGCCTGAC
AGCTGAGAACGCAGCTGTATTTTACTGTGCGAGCGTTAAATGGGGATATAGT
GGCCACGAATACTCAGACTTCCAGAACTGGGGCCAGGTCACCCTGGTCATCG
TCTCCTCAG

La première ligne, comme dans un fichier .fasta classique, contient l'identifiant de la séquence. En revanche, dans le cas des simulations, nous avons des informations supplémentaires séparées par un "|". De gauche à droite nous avons, l'identifiant d'un gène V (IGHV3-30-3*01), l'identifiant d'un gène D (IGHD5-12*01), l'identifiant d'un gène J (IGHJ1*01|40) et un nombre entier. Les identifiants de gène correspondent au gène présent dans le modèle que l'on est censé détecter, ce sont ces étiquettes qui nous permettent de comparer nos résultats et voir si l'on a un vrai positif ou un faux positif. Le nombre, quant à lui, est un paramètre à fixer lorsque l'on génère un set de simulation. Il s'agit du nombre de mutations aléatoires voulues dans la simulation. En fixant ce nombre, nous obtenons ainsi des séquences simulées avec des mutations éparpillées au hasard tout le long. Dans notre exemple, le nombre est 40 ; nous avons donc 40 mutations aléatoires dispersées le long de la séquence.

3.3 Etapes de l'algorithme

3.3.1 Identification des gènes V et J

3.3.1.1 Premier résultat

La première étape de l'algorithme est une fonction qui consiste à diviser les séquences en des kmers. Le but est ainsi de comparer les kmers de la séquence query avec tous les kmers des différentes séquences références. La première version de cette fonction renvoie une liste avec tous les kmers de taille k. Cette méthode s'est avérée mauvaise avec le temps. En effet, en comparant deux listes de kmers, nous avons observé que des kmers formés au début de la 1ère séquence se comparent avec des kmers formés à la fin de la 2ème séquence, ce qui falsifie donc le résultat. La démarche des kmers doit alors être plus précise. Nous avons donc décidé d'associer un kmer avec un indice pour la deuxième version. De cette manière, chaque groupe successif de kmer possède un indice propre à lui. Par exemple, pour la séquence Seq1 : "CGACAAAC", k : 3 et le nombre d'indice : 2, la fonction kmer divise cette séquence de la manière suivante :

$\text{Seq}_k mer1 = [['CGA', 0], ['GAC', 0], ['ACA', 1], ['CAA', 1], ['AAA', 2], ['AAC', 2]]$

Un kmer d'un indice 0 ne peut alors pas se comparer avec un kmer d'indice 2. Ainsi, la comparaison de deux séquences en les divisant en des kmers avec indices, garantit une comparaison rapide et efficace.

La deuxième étape de l'algorithme est une fonction qui compare les [kmer, indice] de la séquence query avec ceux des séquences de référence. L'algorithme commence tout d'abord par comparer les V puis les J et enfin les D. Mais un problème se pose. Toute la séquence

query est divisée en [kmer, indice]. Ainsi, les [kmer, indice] supposés être pour les D ou J, sont mélangés avec ceux de V. Afin de résoudre ce problème, nous avons fixé un seuil pour les V et les J. Le seuil est la taille moyenne de chaque partie. Il représente ainsi 280 nucléotides pour les V et 50 nucléotides pour les J. De la sorte, l'algorithme s'arrête une fois qu'il atteint le seuil de la partie concernée. Cette méthode s'est avérée utile mais peu optimale. En effet, le temps d'exécution pour la partie V sur 200 séquences est trop lent (Figure 3.2). L'algorithme prend environ 4 minutes et 42 secondes pour la comparaison de seulement 200 séquences. En revanche, le pourcentage de vrais positifs est plus ou moins satisfaisant (82.5% pour le data set contenant 40 mutations) (Figure 3.2).

```
--------------------------------------------------------------------------------
                                Résultat final
--------------------------------------------------------------------------------
Nom du fichier ref                          : human_germline//IGHV.fasta
Nom du fichier query                        : simple_plus_indels_40Mut_out.fasta
Nombre de vrais positifs                    : 165
Nombre de séquences testées                 : 200
Pourcentage de vrais positifs pour les V    : 82.5 %
Temps final de l'algorithme                 : 00 04 46
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

FIGURE 3.2 – Résultat de l'algorithme sur la partie V avec 200 séquences (de 40 mutations chacune) étudiées et un seuil de 280 nucléotides.

3.3.1.2 Deuxième résultat

Une hypothèse sur le temps s'est alors émise : certains V ont des tailles beaucoup plus petites que la moyenne et l'algorithme continue ainsi d'analyser des [kmer, indice] inutiles avant d'atteindre le seuil fixé. Afin de prouver cette hypothèse, nous avons décidé de retirer le seuil pour les V et ajouter à la place une variable de blocage. En effet, la variable de blocage compte le nombre de fois successive où aucun [kmer, indice] dans la liste de la séquence query n'a été trouvée dans la liste de la séquence référence. De cette manière, la fonction arrête de réaliser la comparaison si le nombre de blocage est atteint et mémorise la position du dernier [kmer, indice] qui a pu être trouvé. Par exemple, prenons la même séquence Seq1 d'avant qui aura le rôle de séquence de référence :

$\text{Seq}_k mer1 = [['CGA', 0], ['GAC', 0], ['ACA', 1], ['CAA', 1], ['AAA', 2], ['AAC', 2]]$

Supposons qu'on a une deuxième séquence Seq2 qui aura pour rôle la séquence query que l'on cherche à identifier et qui possède aussi une liste de [kmer, indice] :

$\text{Seq}_k mer2 = [['CGA', 0], ['GAC', 0], ['ACA', 1], ['CAT', 1], ['ATT', 2], ['TTC', 2]]$

Nous représentons en rouge la différence des deux listes. Supposons maintenant que la variable de blocage soit égale à 2. Ainsi, en comparant les deux listes, la fonction va s'arrêter sur la 5ème position des listes, soit ['ATT', 2] de la $\text{Seq}_k mer2$.

Blocage : $Blocage1, Blocage2$

$\text{Seq}_k mer1 = [['CGA', 0], ['GAC', 0], ['ACA', 1], ['CAA', 1], ['AAA', 2], ['AAC', 2]]$
$Seq_k mer2 = [['CGA', 0], ['GAC', 0], ['ACA', 1], ['CAT', 1], ['ATT', 2], ['TTC', 2]]$

Alors, la fonction va mémoriser la position du dernier [kmer, indice] qui a pu être trouvé dans les deux séquences soit ['ACA', 1] se situant à la 3ème position.

Ainsi, la fonction renvoie la dernière position de la séquence V suivant la formule suivante :

```
position = k + len(query_kmer[0:dernier_kmer_indice])
```

Avec :
k : taille des kmers
query kmer : liste des [kmers, indice] de la séquence query
dernier kmer indice : position du dernier [kmer, indice] qui a pu être trouvé

Dans notre cas,

```
position = 3 + len(query_kmer[0:3])
position = 3 + 2
position = 5
```

Après plusieurs tests, nous en avons déduit que le nombre optimal d'indice pour les V est de 7 et le nombre optimal de blocage est de 35.

En exécutant l'algorithme, nous trouvons les résultats de cette méthode satisfaisants. En effet, le temps de l'algorithme a pu être réduit de 64%, passant ainsi de 4 minutes 42 secondes à 1 minute 41 secondes pour un même pourcentage de vrais positifs (83.5%) (Figure 3.3).

```
--------------------------------------------------------------------------------
                                Résultat final
--------------------------------------------------------------------------------
Nom du fichier ref                          : human_germline//IGHV.fasta
Nom du fichier query                        : simple_plus_indels_40Mut_out.fasta
Nombre de vrais positifs                    : 167
Nombre de séquences testées                 : 200
Pourcentage de vrais positifs pour les V    : 83.5 %
Temps final de l'algorithme                 : 00 01 41
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

FIGURE 3.3 – Résultat de l'algorithme sur la partie V avec 200 séquences (de 40 mutations chacune) étudiées et avec allèle.

3.3.1.3 Troisième résultat

Notre deuxième objectif est d'optimiser le pourcentage de vrais positifs de l'algorithme du projet. Une hypothèse s'est mise en place : il existe des séquences références qui sont très proches et, en les comparant avec notre séquence query, l'algorithme du projet renvoie donc le même score pour ses séquences références. Afin de prouver cette hypothèse, nous avons modifié notre l'algorithme pour qu'il nous renvoie le nom de toutes les séquences références de même score. À notre surprise, on observe qu'il existe des séquences query avec plusieurs séquences références possibles (Figure 3.4). En regardant plusieurs tests, on s'aperçoit que la majorité des séquences références avec un même score trouvé sont bien de mêmes gènes mais d'allèles différentes. Après plusieurs recherches, nous en avons conclu que ces séquences trouvées pouvaient être considérées comme étant identiques. En effet, les allèles sont représentées à la fin du nom de la séquence par le signe '*' suivi d'un nombre. L'algorithme du projet supprime alors le signe '*' et tout ce qui se trouve

après des étiquettes pour que ces cas deviennent un hit et ne soient pas considérés comme des faux négatifs. Par exemple, les séquences avec le même score de la Figure 3.4 sont maintenant tous considérés comme les mêmes.

```
--------------------------------------------------------------------------------
Sequences avec même score : ['IGHV3-33*01', 'IGHV3-33*05', 'IGHV3-33*06']
Nombre d'iteration        : 179
Sequence reference        : IGHV3-33*05
Sequence query            : IGHV3-33*01
Temps levenshtein         : 3.0994415283203125e-05
Temps complet:            : 1.5318341255187988
-> FAUX Différente sequence trouvée
--------------------------------------------------------------------------------
```

FIGURE 3.4 – Une séquence query avec plusieurs séquences références de même score.

Désormais, en exécutant de nouveau l'algorithme du projet, on observe une amélioration de 13.5% du pourcentage de vrais positifs. En effet, le pourcentage de vrais positifs est passé de 83.5% à 97.0% ; une amélioration très considérable et qui rend donc l'algorithme du projet plus précis (Figure 3.5)

```
--------------------------------------------------------------------------------
                                Résultat final
--------------------------------------------------------------------------------
Nom du fichier ref                        : human_germline//IGHV.fasta
Nom du fichier query                      : simple_plus_indels_40Mut_out.fasta
Nombre de vrais positifs                  : 194
Nombre de séquences testées               : 200
Pourcentage de vrais positifs pour les V  : 97.0 %
Temps final de l'algorithme               : 00 01 41
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

FIGURE 3.5 – Résultat de l'algorithme sur la partie V avec 200 séquences (de 40 mutations chacune) étudiées avec suppression des allèles.

Suivant le même principe, nous avons trouvé qu'il existe des séquences références avec le même score trouvée qui se différencie par le symbole D (Figure 3.6).

```
Sequences avec même score : ['IGHV3-23*04', 'IGHV3-23D*02']
Nombre d'iteration        : 150
Sequence reference        : IGHV3-23D
Sequence query            : IGHV3-23
Temps levenshtein         : 1.7881393432617188e-05
Temps complet:            : 0.585881233215332
-> FAUX Différente sequence trouvée
```

FIGURE 3.6 – Séquences avec un même score qui se différencie par la lettre 'D'.

Après avoir fait des recherches, nous avons trouvé que le symbole D représente la version dupliquée du gène. En suivant cette idée, le gène "IGHV3-23D*02" de notre exemple correspond à la version dupliquée de "IGHV3-23*02". Ainsi, nous avons émis l'hypothèse que ces deux séquences sont quasi-identiques et peuvent donc être considérées comme un "hit". Afin de prouver cette hypothèse, nous avons réalisé plusieurs alignements sur ces cas-là avec l'outil ClustalW. De la sorte, nous avons constaté que nous ne pouvons pas faire la distinction entre ces deux gènes (Figure 3.7). Nous avons donc considéré ces deux gènes comme étant identiques.

FIGURE 3.7 – Alignement des gènes “IGHV3-23*02” et “IGHV3-23D*02” avec l’outil Clustal.

Ainsi, l’algorithme du projet a subi une petite amélioration de 1%, passant de 97% à 98% pour un même temps d’exécution.

Résultat final de V sur 40 mutations

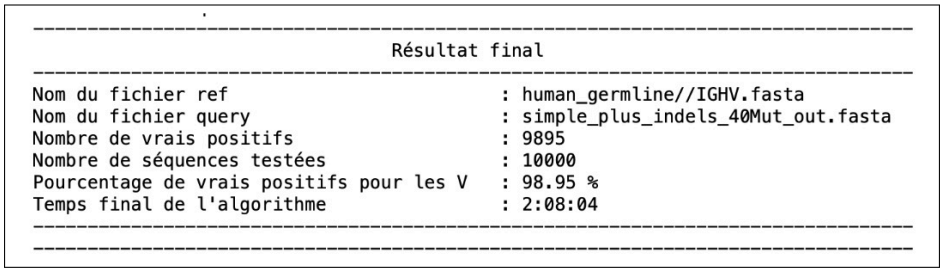

FIGURE 3.8 – Résultat final de V sur 10 000 séquences et 40 mutations.

Le segment J suit exactement la même démarche que le segment V mais en parcourant les séquences à l’envers. Son implémentation a donc été assez rapide. Après plusieurs tests, nous en avons déduit que le nombre optimal d’indice pour les J est de 10 et le nombre optimal de blocage est de 20.

Résultat final de J sur 40 mutations

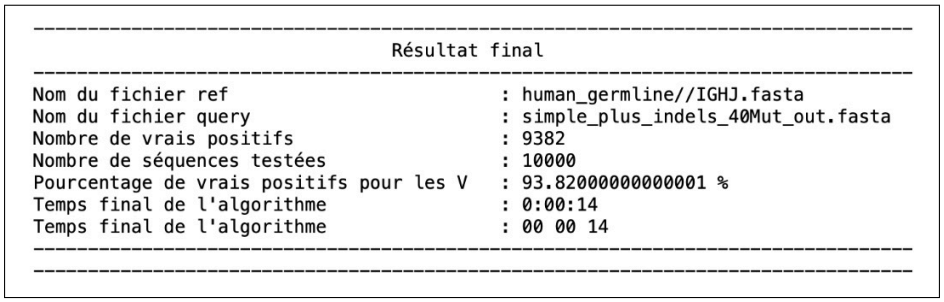

FIGURE 3.9 – Résultat final de J sur 10 000 séquences et 40 mutations.

3.3.2 Identification des gènes D

Après avoir fini les segments V et J, l'algorithme du projet continue avec le dernier segment, le segment D. Les segments D sont la partie la plus difficile des séquences. En effet, les segments sont très petits et diffèrent énormément en taille entre une séquence et une autre. Nous pouvons trouver des tailles de 16 nucléotides comme nous pouvons trouver des tailles de 31 nucléotides (Figure 3.10) rendant plus difficile d'identifier le gène en particulier. Ce n'est pas tout, en effet les gènes D sont séparés des V et J par des blocs variables de nucléotides. Ces blocs sont également sujet à des insertions ou délétions et ont une taille encore plus variable que les gènes eux-mêmes. Ainsi, la précision du bloc D repose sur la précision des blocs V et J dans notre cas. Il devenait donc impossible de continuer uniquement sur la méthode des [k-mer, indice] suivant notre implémentation étant donné qu'un bloc D ne commence quasiment jamais par la région de son gène. Déterminer la région spécifique où commençait le gène D est une tâche quasiment impossible. Pour cette raison, après quelques recherches et consultations avec notre encadrante, nous avons décidé d'employer la méthode des Hidden Markov Models (HMM) qui semble être la méthode produisant les résultats les plus satisfaisants sur les D.

```
Nom séquence    : IGHD3/OR15-3b*01
Séquence        : gtattatgatttttggactggttattatacc
Taille séquence : 31 nt
------------------------------------------------
Nom séquence    : IGHD4-11*01
Séquence        : tgactacagtaactac
Taille séquence : 16 nt
```

FIGURE 3.10 – Taille de deux séquences D.

Ainsi, l'analyse sur les séquences D doit être plus sensible et délicate que celle réalisée sur les V et J.

3.3.2.1 Premier résultat

La première démarche que nous avons effectué est de tester directement l'algorithme du projet sur les D. Nous avons émis l'hypothèse que l'algorithme du projet ne va pas émettre de bons résultats à cause de la difficulté de D pour les raisons déjà citées. Ainsi, en testant l'algorithme du projet, nous avons bien obtenu des résultats insatisfaisants. En effet, pour 200 séquences et 40 mutations, l'algorithme du projet renvoie 30% de vrais positifs (Figure 3.11).

```
--------------------------------------------------------------------------------
                              Résultat final
--------------------------------------------------------------------------------
Nom du fichier ref                        : human_germline//IGHD.fasta
Nom du fichier query                      : simple_plus_indels_40Mut_out.fasta
Nombre de vrais positifs                  : 60
Nombre de séquences testées               : 200
Pourcentage de vrais positifs pour les V  : 30.0 %
Temps final de l'algorithme               : 0:00:00
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

FIGURE 3.11 – Premier résultat de D pour 200 séquences de 40 mutations.

Nous avons donc émis une autre hypothèse concernant ce faible taux de positivités : les séquences J, aussi précises qu'elles soient, prennent des nucléotides qui sont propres à D et ainsi rendent le résultat de D mauvais. Alors, nous avons analysé les séquences de D et réalisé plusieurs tests. Nous avons ainsi constaté que le pourcentage de taux positifs de D augmente de peu (+1,5%) en prenant 9 nucléotides du début de la séquence de J. Cette augmentation ne peut pas être considérée comme une bonne amélioration et donc une nouvelle approche sur les séquences de D doit avoir lieu.

3.3.2.2 Deuxième résultat

Notre encadrante nous a donc proposé d'utiliser les HMM. Elle nous a mentionné notamment une publication qui indique que la méthode des HMM produit les meilleurs résultats pour détecter les D. En effet, un HMM peut représenter une séquence sous la forme d'états match, deletion et insertion. En représentant une séquence sous cette structure, il devient bien plus facile de détecter les vrais positifs en prenant en compte les scénarios de mutation.

Pour pouvoir travailler avec des HMM, il faut en premier lieu le créer en lui fournissant des données d'entraînement. Ainsi pour créer un HMM pour les D, il faut récupérer des séquences de gènes D identifiés via laboratoire comme étant véridiques, les aligner puis finalement construire le HMM via cet alignement. Nous avons utilisé une base de données nommée OAS qui nous a proposé un grand nombre de jeux de données avec des gènes D annotés. Pour filtrer les données à utiliser, nous avons sélectionné les champs organism et disease et avons coché respectivement human et SARS-COV-2 dans la catégorie unpaired sequences. Les données étaient stockées dans des fichiers .csv, ces données ont ensuite été extraites puis stockées dans des fichiers .fasta prêts à être alignés. Parmi les données, nous avons pu retrouver un certain nombre de séquences identiques, les doublons ont été supprimés pour plus de confort.

Pour avoir un échantillon d'entraînement convenable, notre encadrante nous avait conseillé d'essayer de tabler sur environ 100 séquences minimum par gène D de référence. Cependant, malgré le volume important de données et vu qu'elles étaient centrées sur une maladie en particulier, certains gènes ont dû se contenter d'environ 80 séquences d'entraînement. En moyenne, nous nous sommes donc retrouvés avec 266 séquences d'entraînement par gène D. Une fois les alignements effectués nous avons donc pu construire les HMM.

Désormais, étant donné que nous avons maintenant des alignements pour chaque D, nous pouvons commencer à appliquer HMM. Nous avons en premier lieu utilisé un logiciel nommé HMMER qui est un script qu'on peut exécuter sur bash et qui nous construit un HMM à partir d'un alignement de séquences. À noter que la version utilisée pour HMMER est "hmmer-3.1b2". La 1ére étape est de créer un fichier avec la commande hmmbuild qui nous permet de créer les modèles de chaque alignement de D des séquences références (Figure 3.12).

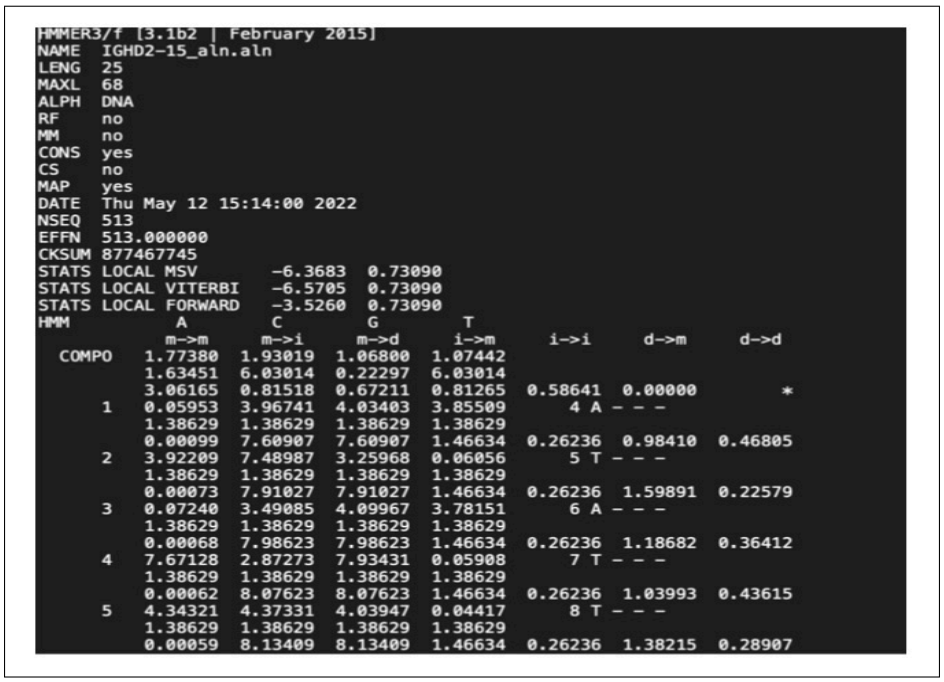

FIGURE 3.12 – Exemple de modèle créer avec la command hmmbuild pour la séquence 'IGHD2-15'.

Une fois les modèles créés, nous analysons chaque séquence query D avec tous les modèles créés. En effet, nous avons extrait toutes les séquences query D avec l'algorithme du projet et nous les avons sauvegardées dans un fichier. Ainsi, le but est de trouver le modèle qui renvoie la meilleure E-value de chaque séquence query extraite. En effet, le E-value est un paramètre qui décrit le nombre de 'hits' qu'on peut s'attendre à voir par hasard quand on cherche dans une base de données. E-value diminue exponentiellement quand le score augmente. Alors plus la valeur de E-value est petite ou proche de zéro, plus le match est important. Pour faire cela, nous avons créé un deuxième fichier qui nous permet de faire cette procédure avec la commande hmmsearch. Ainsi, nous obtenons un fichier qui renvoie les meilleurs 'hits' pour chaque séquence query D (Figure 3.13).

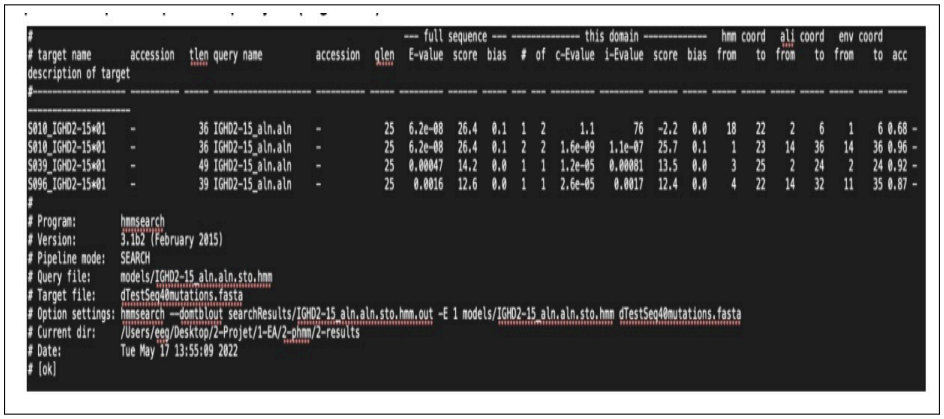

FIGURE 3.13 – Exemple de résultat obtenu avec la command hmmsearch pour la séquence 'IGHD2-15'.

La troisième étape est d'exécuter une 3ème commande bash de HMMER qui nous permet de combiner tous nos résultats dans un seul fichier texte. Enfin, nous avons écris une

fonction qui nous permet d'analyser le fichier texte, de trier les séquences suivant leurs E-value et de calculer le taux de true positifs (TP), false positifs (FP) et false négatifs (FN) de toutes les séquences D query étudiées. Ainsi, nous obtenons les résultats suivants : (Figure 3.14)

```
--------------------------------------------------------------------------------
                                Résultat final
--------------------------------------------------------------------------------
Nom du fichier ref                          : Séquence D extraites
Nom du fichier query                        : simple_plus_indels_40Mut_out.fasta
Nombre de vrais positifs                    : 44
Nombre de false positifs                    : 1
Nombre de false negatifs                    : 156
Nombre de séquences testées                 : 200
Pourcentage de vrais positifs pour les D    : 22.0 %
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

FIGURE 3.14 – Nombre de TP, FP et FN des séquence D étudiées.

Nous remarquons que malheureusement nous n'avons pas pu augmenter le nombre de positifs de D. Le pourcentage de vrais positifs est passé de 30% à 22%, une diminution considérable par rapport à la 1ère version. Cependant, nous avons remarqué un point très important. L'algorithme HMM renvoie les bons résultats pour les séquences auxquelles il a pu attribuer un modèle (VP = 44). Les faux positifs sont quasi-inexistants (FP = 1). Ainsi, la majorité des modèles de faux négatifs sont des séquences auxquelles HMM n'a pas pu trouver un modèle propre à eux (FN = 156). Alors nous avons eu l'idée de prendre ces séquences-là et de réaliser sur eux l'algorithme du projet initial. Ainsi, nous combinons l'algorithme du projet avec HMM.

3.3.2.3 Troisième résultat

La construction de HMM et la recherche des séquences query avec les modèles HMM sont réalisées avec les commandes bash comme décrit précédemment et non sur python. Or l'algorithme du projet initial est codé sur python. La combinaison des deux méthodes pose ainsi une contrainte. Nous avons alors fait des recherches pour trouver une méthode qui nous permet de réaliser HMM directement sur python. C'est ainsi que nous avons trouvé la librairie 'pyhmmer'. L'utilisation de la librairie était extrêmement difficile et contre-intuitive. Contrairement au HMM 'normal', la libraire exige que la taille de la séquence query soit de la même taille que la séquence référence. De la sorte, nous étions obligés de modifier la taille de notre séquence query afin qu'elle soit identique à celle de la séquence référence. Ainsi, trois possibilités se posent :

1. Les deux séquences query et références sont de mêmes tailles alors aucune modification n'est nécessaire;
2. La séquence query a une taille inférieure à celle de la référence. Dans ce cas, nous ajoutons des gaps dans la séquence query;
3. La séquence query a une taille supérieure à celle de la référence. Dans ce cas, nous divisons notre séquence query en trois sous-séquence de tailles égales à celle de références.

Donnons un exemple pour le cas 3. Considérons la séquence query suivante : "AACCTTGG" de taille 8. De même, considérons la séquence référence de taille 6. Ainsi, notre fonction renvoie,

1. `Sous-séquence 1:` AACCTT
2. `Sous-séquence 2:` CCTTGG
3. `Sous-séquence 3:` ACCTTG

Nous obtenons alors trois séquences de taille égales à celle de références. La première sous-séquence a été obtenue en commençant par le début de la séquence query, jusqu'à atteindre la taille la séquence référence. La deuxième sous-séquences a été obtenue en commençant par la fin de la séquence query, jusqu'à atteindre la taille la séquence référence. La troisième séquence a été trouvée en enlevant à tour de rôles, un nucléotide au début puis un nucléotide à la fin de la séquence query jusqu'à atteindre la taille de la séquence référence. Nous réalisons cette procédure pour chaque séquence query étudiée et sur chaque modèle. Une fois que nous avons les trois sous-séquences dans un fichier, nous pouvons réaliser le "hmm search" sur python avec la fonction de la libraire "pyhmmer" suivante :

```
hits = pipeline.search_hmm(query=hmm_file.read(), sequences=sequences).
```

Puis, nous nous assurons qu'il existe bien un hit avec la fonction de la libraire "pyhmmer" suivante :

```
ali = hits[h].domains[d].alignment
```

Cependant, une séquence query peut avoir plusieurs hits avec différentes séquences références (Figure 3.15).

```
                  msa   4 TAGTGG 9
                          A TGG
IGHD5-12_aln.hmm1   5 AAATGG 10
                          344444 PP

                  msa   1 ATATAGTGGCTACGATT 17
                          ATATAGTGGC+ACGA T
IGHD5-12_aln.hmm1  13 ATATAGTGGCCACGAAT 29
                          9**************87 PP

                  msa   1 ATATAGTGGCTACGA 15
                          ATA+   G CT C+A
IGHD5-12_aln.hmm1  28 ATACTCAGACTTCCA 42
                          899999*******99 PP
```

FIGURE 3.15 – Différents hits pour une même séquence query.

Ainsi, nous calculons le score des différents hit grâce à la fonction de la libraire "pyhmmer" suivante :

```
score = hits[h].domains[d].score
```

De la sorte, nous préservons le hit avec le meilleur score et nous renvoyons le nom de sa séquence de référence. Enfin, nous testons si nous avons bien eu la séquence voulue avec HMM. Comme déjà décrit, les séquences qui ont trouvé des hits ont été bien classifiées. Nous nous intéressons donc aux séquences sur lesquelles nous n'avons pas trouvé un hit.

Nous appliquons donc l'algorithme du projet sur ces séquences. À notre surprise, le taux de positifs a pu être augmenté de 14%, une augmentation très considérable et importante pour la partie D de 40 mutations (Figure 3.16).

Résultat final de D sur 40 mutations

```
--------------------------------------------------------------------------------
                                Résultat final
--------------------------------------------------------------------------------
Nom du fichier ref                          : human_germline//IGHD.fasta
Nom du fichier query                        : simple_plus_indels_40Mut_out.fasta
Nombre de vrais positifs                    : 4472
Nombre de vrais positifs sans HMM           : 1050
Nombre de faux positifs                     : 1341
Nombre de séquences non trouvées            : 4047
Nombre de séquences testées                 : 10000
Pourcentage de vrais positifs pour les D    : 44.72 %
Temps final de l'algorithme                 : 0:00:09
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

FIGURE 3.16 – Résultat trouvé pour la partie D en utilisant l'algorithme du projet combiné avec HMM.

Chapitre 4

Comparaison des résultats de l'algorithme du projet avec Dsab

Avant de commencer la comparaison, nous aimerions souligner quelques points imortants sur l'algorithme DSab. Dans la publication de DSab, nous pouvons retrouver des performances satisfaisantes de l'algorithme. En revanche, les observations que nous avons effectuées en exécutant le leur semblent être en contradiction. En effet, les performances pour les gènes D et J sont très imprécises voire aléatoires pour 200 séquences. Nous avons obtenu un résultat de 1% pour les D et un résultat d'environ 29% pour les J (Figure 4.1). Les tests pour la performance des V n'ont cependant pas pu être effectués étant donné le temps très long d'exécution de DSab pour ces gènes (183 sec * 200 séquences = 36 600 sec, environ 10h). Toutefois nous obtenons une performance de 100% sur 10 séquences.

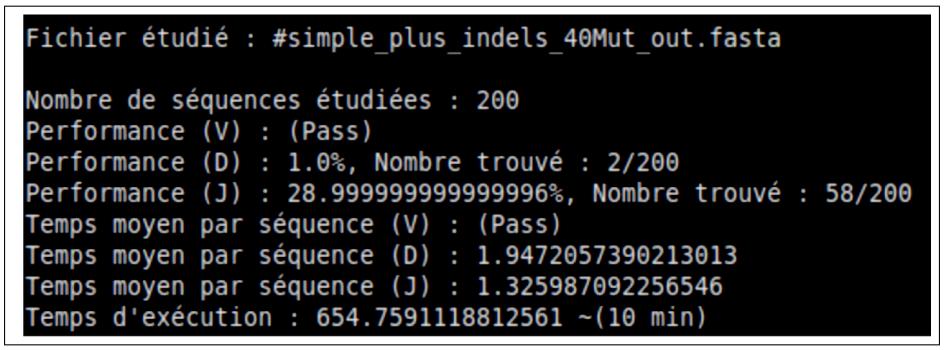

FIGURE 4.1 – Résultat trouvé pour l'exécution de DSab.

Dans tous les cas, nous avons réalisé la comparaison de notre algorithme avec les résultats fournis dans la publication de DSab.

4.0.0.1 Comparaison du pourcentage de vrais positifs

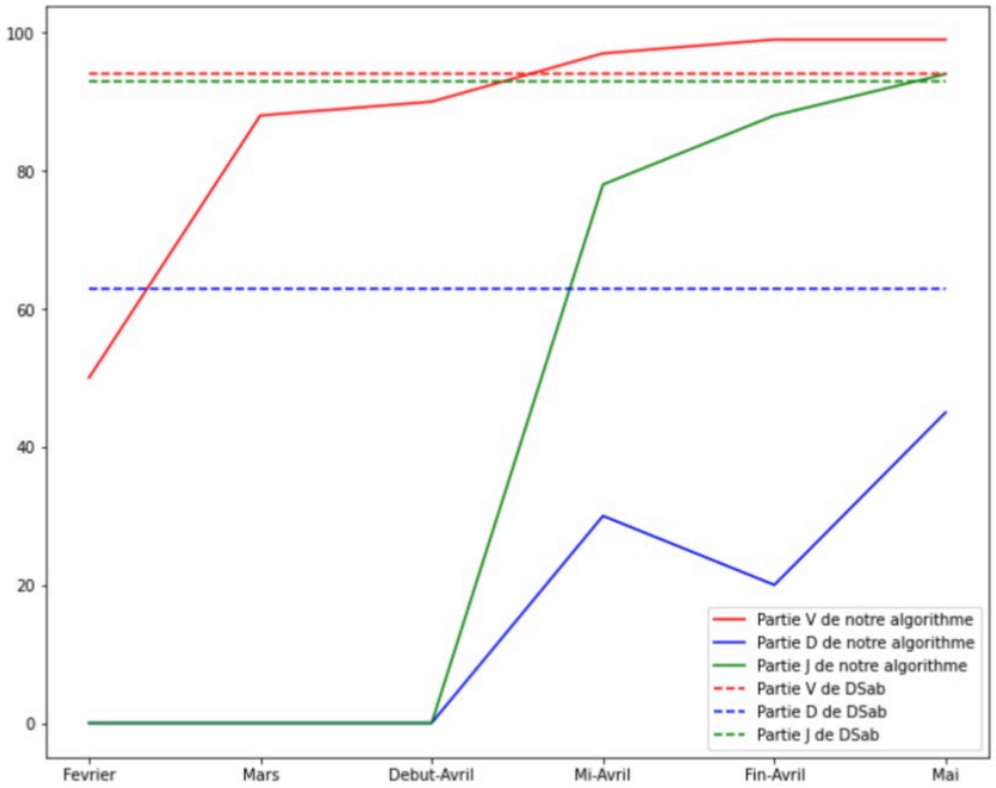

FIGURE 4.2 – Comparaison des taux de positifs des V, D et J entre l'algorithme du projet et DSab sur les séquences de 40 mutations.

Les graphes ci-dessous montrent les performances de l'algorithme du projet au cours du temps (Figure 4.2). Nous remarquons que la partie qui nous a pris le plus de temps est la partie V. C'est la partie avec laquelle nous avons commencé le projet, la partie où nous étions encore en train de faire les premières recherches et avoir les premiers aperçus du sujet. Nous avons voulu améliorer le plus que possible le pourcentage de vrais positifs de V pour avoir les résultats les plus précis. Et c'est ainsi que nous avons non seulement dépassé DSab pour les V (97 % pour l'algorithme du projet contre 94% DSab en Mi-Avril) mais nous avons aussi pu améliorer l'algorithme du projet à 98.95% pour le pourcentage de vrais positifs à la fin du projet. La deuxième partie du projet était l'étude de la partie J. L'étude était rapide avec de petites modifications faites de temps à autre. À la fin du projet, nous avons aussi pu dépasser le pourcentage de vrais positifs pour la partie V de DSab. Nous avons obtenu un pourcentage de vrais positifs de 93.82% contre 93.51% pour DSab. Enfin, la partie D était la dernière partie que nous avions traité. C'est la partie avec laquelle nous avions eu un temps assez limité afin que nous puissions l'optimiser le plus précisément possible. Cependant, nos résultats restent très positifs considérant le temps limité que nous avions. Un pourcentage de vrais positifs de 44.72% pour 40 mutations contre 62.89% de DSab en une courte durée montre le grand potentiel que l'algorithme du projet possède. Nous sommes très optimistes qu'avec plus de temps, l'algorithme du

projet sera non seulement meilleur que DSab sur les parties V et J mais aussi sur la partie D.

4.0.0.2 Comparaison du temps d'exécution

	V	D	J
DSab	182.75	1.77	1.200000
Notre Algorithme	0.53	0.03	0.000018

FIGURE 4.3 – Temps d'exécution en seconde et sur 40 mutations d'une seule séquence en utilisant l'algorithme du projet et DSab.

L'algorithme du projet a aussi un temps d'exécution largement meilleur à celui de DSab (Figure 4.3). En effet, la méthode des [kmers, indice] utilisée dans l'algorithme du projet s'est avérée largement meilleure en performance et en temps d'exécution que la méthode BLAST utilisée pour les parties V et J de DSab.

Chapitre 5

Conclusion

Durant ce projet, nous avons pu découvrir comment identifier et annoter des gènes sur une séquence et les procédés qui accompagnent un tel travail. Nous avons pu voir que pour accomplir une telle tâche, il fallait passer par des étapes très diversifiées selon les gènes recherchés qu'il s'agisse de l'algorithme de [kmer, indice] que nous avions développé ou bien les HMM. Comprendre les spécificités de chaque type de gène tels que leurs positions et leurs tailles afin de les exploiter pour obtenir des résultats est une faculté importante. Il est clair que la complexité du projet demandait une attention considérable de notre part. La bonne maîtrise des multiples, énormes et divers fichiers, demandait une parfaite gestion et organisation de notre part. Tout au long du semestre, nous avons bien eu le temps de contrôler cette méthodologie de recherche de gènes et avons obtenu des performances très satisfaisantes sur l'algorithme du projet. Ainsi nous avons pu obtenir de meilleurs résultats sur les V et les J que DSab, l'un des plus puissants algorithmes d'identification de gènes, et des meilleurs temps d'exécution sur toute la séquence, un exploit dont nous sommes très fiers. Nous sommes donc très optimistes sur le potentiel de l'algorithme du projet, un algorithme qui pourra un jour analyser des séquences issues de données réelles et contribuer à des avancées scientifiques et médicales. Nous espérons ainsi que les nouvelles générations de scientifiques de Sorbonne Université pourront continuer à perfectionner l'algorithme du projet et le rendre encore plus optimal et performant.

Chapitre 6

Bibliographie

[1] Shrikumar Avanti, Prakash Eva, and Kundaje Anshul. 2019. GkmExplain : fast and accurate interpretation of nonlinear gapped k-mer SVMs. Bioinformatics 35, 14 (2019), i173–i182. DOI :https ://doi.org/10.1093/bioinformatics/btz322

[2] Brown C. Titus and Irber Luiz. 2016. An introduction to k-mers for genome comparison and analysis — sourmash 4.2.5.dev10+gefebe8a1.d20220304 documentation. Retrieved April 3, 2022 from https ://sourmash.readthedocs.io/en/latest/kmers-and- minhash.html

[3] Taylor John Shawe and Cristianini Nello. 2004. Kernel Methods for Pattern Analysis. Cambridge University Press.
Retrieved April 3, 2022 from https ://doi.org/ 10.1017/CBO9780511809682

[4] Samocha Kaitlin E. 2014. A framework for the interpretation of de novo mutation in human disease. Nat. Genet. 46, (2014), 944–950. DOI :https ://doi.org/10.1038/ ng.3050

[5] Compeau P., Pevzner P., and Teslar G. 2011. How to apply de Bruijn graphs to genome assembly. Nat. Biotechnol. 29, (2011), 987–991. DOI :https ://doi.org/10.1038/ nbt.2023

[6] Meinicke Peter, Tech Maike, Morgenstern Burkhard, and Merkl Rainer. 2004. Oli- go kernels for datamining on biological sequences : a case study on prokaryotic transla- tion initiation sites. BMC Bioinformatics (2004).

[7] Qingchen Zhang, Lu Zhang, Zhou Chen, Yang Yiyan, Yin Zuojing, Wu Dingfeng, Tang Kailin, and Cao Zhiwei. 2019. DSab-origin : a novel IGHD sensitive VDJ mapping method and its application on antibody response after influenza vaccination. BMC 4 Sorbonne Université - M1 INFORMATIQUE Bioinformatics (2019), 0–9.
DOI :https ://doi.org/10.1186/s12859-019-2715-7

[8] Chen Zhen, Zhao Pei, Li Chen, Li Fuyi, Xiang Dongxu, Chen Yong-Zi, Akutsu Tatsuya, Daly Roger J, Webb Geoffrey I, Zhao* Quanzhi, Kurgan* Lukasz, and Song* Jiangning. 2021. iLearnPlus : a comprehensive and automated machine-learning plat- form for nucleic acid and protein sequence analysis, prediction and visualization. Nu- cleic Acids Res. (2021). DOI :https ://doi.org/10.1093/nar/gkab122

[9] Chen Zhen, Zhao Pei, Li Fuyi, Leier André, Marquez-Lago Tatiana T, Wang Yanan, Webb Geoffrey I, Smith A Ian, Daly* Roger J, Chou* Kuo-Chen, and Song* Jiangning.

2018. iFeature : a Python package and web server for features extraction and selection from protein and peptide sequences. Bioinformatics (2018), 2499–2502.
DOI :https ://doi.org/10.1093/bioinformatics/bty140

[10] Chen Zhen, Zhao Pei, Li Fuyi, Marquez-Lago Tatiana T, Leier André, Revote Jerico, Zhu Yan, Powell David R, Akutsu Tatsuya, Webb Geoffrey I, Chou* Kuo-Chen, Smith A Ian, Daly* Roger J, Li Jian, and Song* Jiangning. 2020. iLearn : an integrated platform and meta-learner for feature engineering, machine-learning analysis and modeling of DNA, RNA and protein sequence data. Brief. Bioinform. (2020), 1047–1057. Retrieved April 3, 2022 from https ://doi.org/10.1093/bib/bbz041