Université de Sherbrooke

Projet BIN702 2022 - rapport final

Evaluation de méthodes de clustering pour l'homologie

Arthur BAUDOT, Victor MICHON

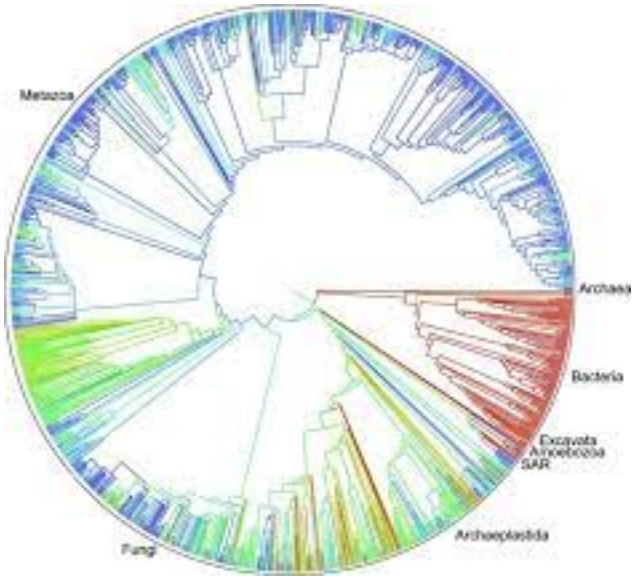


Table des matières

1 Introduction

L'homologie consiste à identifier des liens évolutifs entre les espèces [**?**]. Plus concrètement, on cherche à identifier, entre les espèces, des gènes qui sont très similaires, ce qui met en évidence que celles-ci descendent d'un ancêtre commun. Ce travail est fondamental pour les biologistes, puisqu'il leur permet de retracer l'évolution de toutes les espèces vivantes. Mais celui-ci est très fastidieux, tant les données sont nombreuses et variées.

C'est pourquoi l'informatique peut jouer un rôle dans ce travail, grâce à sa capacité à manipuler rapidement une grande quantité d'informations. De plus, il existe des algorithmes de clustering qui peuvent permettre de regrouper les gènes par similarité. La problématique est alors de savoir quels sont les algorithmes de clustering qui permettent de retracer au mieux et le plus efficacement possible l'évolution des espèces.

Dans ce projet, nous tenterons de répondre à cette problématique, en évaluant différentes méthodes de clustering existantes. Nous étudierons en particulier l'algorithme *UPGMA*, l'algorithme des *K-Means*, ainsi qu'un algorithme de clustering hiérarchique par méthode agglomérative. Nous essaierons également d'implémenter l'algorithme *Transitivity Clustering*, basé sur la structure de graphe. Pour évaluer ces algorithmes, nous simulerons des familles de gènes et nous appliquerons les algorithmes dessus. Nous testerons également les algorithmes sur de vrais gènes, afin de les évaluer sur des cas réels.

Nous commencerons par étudier quelques travaux ayant été réalisés sur le sujet et quelles solutions ont déjà été implémentées. Puis, nous détaillerons comment nous avons procédé pour simuler les familles de gènes et pour écrire les algorithmes de clustering. Nous effectuerons ensuite une analyse des résultats que nous avons obtenus lors du test de nos algorithmes. Ensuite, nous testerons nos algorithmes sur de vrais gènes. Enfin, nous donnerons une réponse à la problématique posée, à savoir donner un ou plusieurs algorithmes répondant au mieux au besoin d'identifier automatiquement des familles de gènes.

2 Revue de littérature

Quatre chercheurs venant du Brésil et de France ont publié un article scientifique en février 2015 [10], nommé *Evaluation and improvements of clustering algorithms for detecting remote homologous protein families* dans lequel ils ont comparé plusieurs méthodes de clustering basés sur la structure de graphe, pour regrouper des protéines par familles, dont nous allons en parcourir trois. Chaque noeud du graphe représente une séquence, et chaque arête entre les noeuds représente la similarité entre les deux séquences reliées.

2.1 Construction de la matrice de similarité

Tous les algorithmes testés dans cette étude se servent d'une matrice de similarité pour construire les familles. Dans cet article, deux façons de déterminer les similarités entre séquences ont été présentées : la comparaison séquence-séquence, et la comparaison profil-profil.

La comparaison séquence-séquence est basée sur le score d'alignement entre deux séquences [10]. On commence par construire un alignement entre chaque paire de séquences à partir de l'algorithme BLAST, qui est une heuristique permettant d'approcher l'alignement optimal. L'algorithme détermine alors un score d'alignement S et une quantité appelée *e-value* ou *expected value*. La *e-value* représente le nombre d'alignements entre les deux séquences qui donnent au moins un aussi bon score que S et qui sont purement dûs au hasard [9]. Ainsi, plus une *e-value* est petite, moins il est probable qu'un alignement de score supérieur ou égal à S soit dû au hasard, et plus il semble que les deux séquences soient réellement proches. Dans la matrice de similarité, on ne stocke pas directement les *e-values* ou les scores S, car ces valeurs dépendent des longueurs des séquences alignées. On calcule et stocke plutôt un score S' pour chaque paire de séquences, indépendant de la longueur des séquences alignées, grâce à la relation suivante [11] :

$$S' = \log_2(\frac{mn}{E}) \tag{1}$$

où m et n sont les longueurs respectives des séquences alignées, et E est la *e-value* associée à l'alignement entre les deux séquences.

La comparaison profil-profil ne se base plus sur l'alignement entre deux séquences, mais sur l'alignement multiple entre plusieurs séquences [3]. Tout comme on le faisait pour deux séquences, on applique une heuristique nommée *HHsearch* qui va déterminer un alignement multiple, et qui va extraire des *e-values* pour chaque paire de séquences de l'alignement multiple obtenu [10]. On a alors juste à appliquer la même transformation que celle de l'équation 1 pour déduire les scores de similarité S'.

2.2 Algorithme 1 : Transitivity Clustering

L'algorithme *Transitivity Clustering* est une méthode de clustering par partition, qui sépare un ensemble de gènes en plusieurs familles de similarité [14]. L'idée globale de cette algorithme est de transformer le graphe initial en un graphe transitif, c'est-à-dire qu'on souhaite avoir un graphe qui est un ensemble disjoint de sous-graphes complets [10]. Chaque sous-graphe complet (aussi appelé *clique*) représenterait alors une famille de gènes. L'algorithme se déroule en deux grandes étapes.

Tout d'abord, on part d'un graphe sans arête, et pour toute paire (i, j) de noeuds, on ajoute une arête entre les deux si leur similarité (obtenue directement par la matrice de similarité) $S'(i, j)$ est supérieure ou égale à un seuil $T \geq 0$. Ainsi, on obtient un graphe dans lequel les séquences très similaires sont reliées entre elles, bien qu'on n'ait pas encore de graphe transitif.

Ensuite, on ajoute et supprime des arêtes itérativement de telle sorte à obtenir une union disjointe de cliques. Un cluster sera alors représenté par une clique. Cependant, on ne modifie pas le graphe n'importe comment. En effet, chaque suppression d'une arête entre i et j a un coût de $S'(i, j)$, et chaque ajout d'une arête entre i et j a un coût de $-S'(i, j)$ [12]. On cherche à minimiser le coût global, qui est égal à la somme de tous les coûts intermédiaires. De ce fait, on pénalisera les suppressions d'arêtes entre des séquences similaires, et les ajouts d'arêtes entre des séquences éloignées. Ce problème d'optimisation est nommé WTGPP (Weighted

Transitive Graph Projection Problem), et est NP-complet [10], c'est-à-dire qu'il n'existe pas d'algorithme trouvant une solution optimale en temps polynômial. Des heuristiques existent pour trouver une solution non optimale mais en temps plus raisonnable, comme par exemple celle expliquée dans l'article [12]. L'idée globale de cette heuristique est de découper le graphe initial G en deux sous-graphes disjoints G_1 et G_2, en supprimant à chaque fois l'arête coûtant le moins cher. Si le coût généré par les suppressions est plus grand que le coût pour effectuer la fermeture du graphe [1], on s'arrête. Sinon, on recommence récursivement le processus avec G_1 puis G_2, jusqu'à ce qu'on ait un coût de suppression qui soit plus grand que le coût de la fermeture du graphe. L'algorithme résultant est exposé en figure 1 [12].

Algorithm REMOVE–CULPRIT(G) returns the highest-scoring edge $\text{argmax}_{uv \in E}\{\Delta_{uv}(G)\}$ and removes it from G. There are m edges; computing each $\Delta_{uv}(G)$ can be done in $O(n)$ since only triples containing uv need to be considered. Thus, the runtime of the first invocation is $O(mn)$. Subsequent invocations need only $O(m+n)$ time, $O(n)$ to update scores for edges around the deleted edge, and $O(m)$ to find the maximum score.

Algorithm TRANSITIVE–CLOSURE–COST(G) assumes G is connected; it returns the total cost of all edge additions required for a transitive closure of G, $\sum_{uv \in \binom{V}{2}} \max\{-s(uv), 0\}$, in time $O(n^2)$.

Algorithm GREEDY–HEURISTIC(G) is the main algorithm. It returns a pair (*deletions*, *cost*), where *deletions* is the list of edges to be removed from G and *cost* is the total cost of all edit operations (both removals and additions). Remember that G is connected.

(1) *cost* ← TRANSITIVE–CLOSURE–COST(G).
(2) If *cost* = 0, return an empty list and cost 0.
(3) Set *deletions* ← *empty list*; *delcost* ← 0.
(4) Repeat the following steps until G consists of two connected components G_1 and G_2.
  (a) uv ← REMOVE–CULPRIT(G)
  (b) append uv to *deletions*
  (c) increase *delcost* by $s(uv)$
(5) Adjust *deletions* such that it only includes edges that contribute to the cut between G_1 and G_2. Adjust *delcost* accordingly, and re-add incorrect edges to G_1 and G_2.
(6) Solve the problem recursively for G_1 and G_2, as long as there is a chance for a better solution:
  If $delcost \geq cost$, return (*empty list*, *cost*).
  $(list_1, cost_1)$ ← GREEDY–HEURISTIC(G_1).
  If $delcost + cost_1 \geq cost$,
    return (*empty list*, *cost*).
  $(list_2, cost_2)$ ← GREEDY–HEURISTIC(G_2)
  If $delcost + cost_1 + cost_2 \geq cost$,
    return (*empty list*, *cost*).
(7) Append $list_1$ and $list_2$ to *deletions*.
  Return (*deletions*, $delcost + cost_1 + cost_2$).

FIGURE 1 – Heuristique permettant d'obtenir un graphe transitif

L'avantage de cette méthode est qu'elle permet d'obtenir un graphe dont on peut extraire très aisément les clusters, puisqu'il suffit de récupérer chaque clique du graphe résultant. En revanche, cet algorithme est difficile et reste assez lourd en temps de calcul. L'initialisation du graphe a une complexité de l'ordre de $O(n^2)$ où n est le nombre de noeuds dans le graphe (puisqu'il faut tester tous les couples (i, j)). L'heuristique permettant d'obtenir un graphe transitif a une complexité de $O(m(m+n)+n^3)$, où m est le nombre d'arêtes dans le graphe et n est le nombre de noeuds dans le graphe [12].

1. La fermeture du graphe correspond aux ajouts d'arêtes qu'il faut faire pour obtenir un graphe transitif.

2.3 Algorithme 2 : Spectral Clustering

Le clustering spectral est une méthode qui exploite les valeurs et les vecteurs propres de la matrice de similarité [10]. L'idée générale est de projeter les données initiales dans un espace généré par les vecteurs propres associés aux valeurs propres les plus prépondérantes, dans lequel les données auront une distribution différente, et d'appliquer un algorithme de clustering dans ce nouvel espace. Dans cet article, les chercheurs appliquent une méthode de clustering spectral adaptée aux protéines, appelée *Spectral Clustering of Protein Sequences* (SCPS) [10].

L'algorithme commence par effectuer une normalisation de la matrice de similarité S, en générant une nouvelle matrice L de la forme [1] :

$$L = D^{-\frac{1}{2}} S D^{-\frac{1}{2}} \tag{2}$$

où D est une matrice diagonale où chaque valeur d_{ii} de D correspond à la somme des scores de similarité entre i et les autres séquences [13] : $d_{ii} = \sum_j s_{ij}$ où s_{ij} est la similarité entre la séquence i et la séquence j.

Ensuite, on diagonalise la matrice L, et on récupère les K colonnes de L correspondant aux K plus grandes valeurs propres de L [13]. On construit alors une matrice U, où chaque colonne de U est un vecteur propre.

Puis, on normalise chaque ligne de U en générant une nouvelle matrice Y, autrement dit, on souhaite que les similarités aient la même unité de mesure. Cette transformation s'effectue de la façon suivante [1] :

$$\forall (i,j), Y_{ij} = \frac{U_{ij}}{\sqrt{\sum_j U_{ij}^2}} \tag{3}$$

Tout ce travail de transformation de la matrice de similarité S a simplement pour objectif de projeter les séquences initiales dans un autre espace. Par la matrice Y, on peut considérer chaque ligne de Y comme un point dans $\mathbb{R}^K$, et chaque ligne de Y représente une séquence.

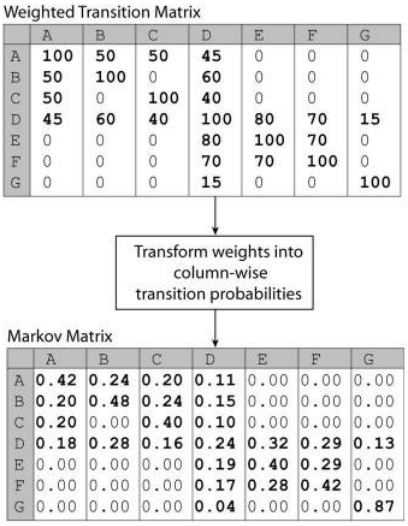
Enfin, on applique l'algorithme des k-means sur la matrice Y, en fixant le nombre de clusters à K [13]. On peut alors construire un graphe où chaque noeud représente une séquence, en liant les noeuds i et j si ceux-ci ont été associés au même cluster [13].

L'avantage de cette méthode est qu'elle est relativement facile à mettre en oeuvre, dans la mesure où il s'agit simplement de transformations de matrices. De plus, une fois la transformation effectuée, on peut déterminer les clusters facilement à l'aide de l'algorithme des k-means. En revanche, cela demande beaucoup d'espace mémoire pour stocker ces différentes matrices, au vu du nombre de séquences. De plus, chaque transformation de matrice demande un temps d'au moins $O(m^2)$, où m est le nombre de séquences du jeu de données, puisque les matrices de similarité sont de taille $m \times m$. Pour contrer ce problème, des opérations de réduction de la taille de la matrice de similarité ont été mises en oeuvre [13] [10].

2.4 Algorithme 3 : Markov Clustering algorithm

L'algorithme de *Markov Clustering* est basé sur le parcours aléatoire dans le graphe. Les arêtes du graphe ne sont plus étiquetées par des similarités, mais par des probabilités. L'idée sous-jacente est qu'il est plus probable qu'il y ait un chemin entre deux noeuds du même cluster qu'un chemin entre deux noeuds de clusters différents, et que les chemins au sein d'un même cluster auront tendance à être plus longs que ceux entre différents clusters [10].

Tout d'abord, on transforme la matrice de similarité en une matrice de probabilité, appelée matrice de Markov. Pour chaque valeur, on a simplement à diviser celle-ci par toutes les autres valeurs de la colonne à laquelle elle appartient, comme cela est illustré sur la figure 2 [2].



Weighted Transition Matrix	A	B	C	D	E	F	G
A	100	50	50	45	0	0	0
B	50	100	0	60	0	0	0
C	50	0	100	40	0	0	0
D	45	60	40	100	80	70	15
E	0	0	0	80	100	70	0
F	0	0	0	70	70	100	0
G	0	0	0	15	0	0	100

Markov Matrix	A	B	C	D	E	F	G
A	0.42	0.24	0.20	0.11	0.00	0.00	0.00
B	0.20	0.48	0.24	0.15	0.00	0.00	0.00
C	0.20	0.00	0.40	0.10	0.00	0.00	0.00
D	0.18	0.28	0.16	0.24	0.32	0.29	0.13
E	0.00	0.00	0.00	0.19	0.40	0.29	0.00
F	0.00	0.00	0.00	0.17	0.28	0.42	0.00
G	0.00	0.00	0.00	0.04	0.00	0.00	0.87

FIGURE 2 – Transformation de la matrice de similarité en une matrice de probabilité

Ensuite, on modifie itérativement la matrice de Markov par deux opérations successives, jusqu'à qu'il n'y ait plus de changements [2]. Ces deux opérations sont l'expansion et l'inflation.

L'expansion de la matrice de Markov M consiste simplement à prendre le carré de la matrice [2]. Ainsi, au fur et à mesure des itérations, on calculera M, puis M^2, puis M^3, etc. L'expansion de la matrice M a pour effet de réduire la longueur des chemins. En effet, chaque valeur de M est comprise entre 0 et 1, donc ces valeurs décroîent au fur et à mesure qu'on augmente la puissance de M.

Après l'expansion, on applique l'inflation, qui consiste à prendre la puissance de Hadamard sur la matrice M, qui correspond à la transformation suivante :[2].

$$\forall (i,j) \in [1,k]^2, \exists r > 1, M_{i,j} = \frac{M_{i,j}^r}{\sum_{l=1}^{k} M_{l,j}^r} \tag{4}$$

Cette opération a tout d'abord pour effet de rendre à nouveau la matrice sous forme stochastique, puis elle augmente les probabilités intra-clusters (au sein des clusters), et diminue les probabilités inter-clusters (entre les clusters) [10].

Enfin, lorsque la matrice M demeure inchangée malgré l'application des deux transformations, on répartit le graphe en sous-graphes disjoints, où chaque sous-graphe correspond à un cluster [10]. L'algorithme de *Markov Clustering* est résumé en figure 3 [2].

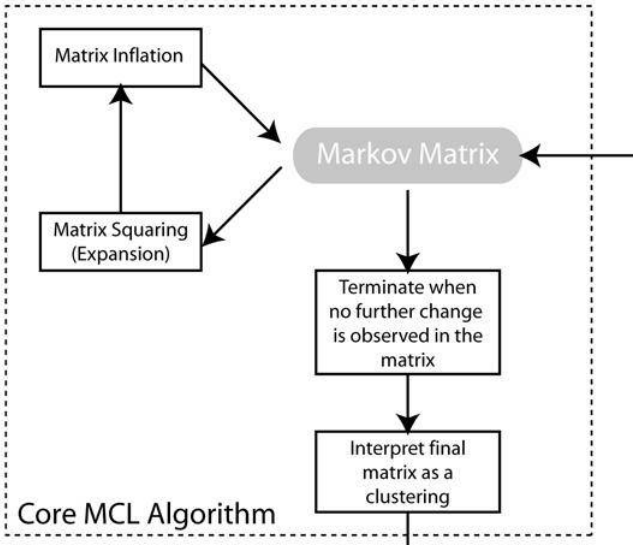


FIGURE 3 – Étapes de l'algorithme de *Markov Clustering*

Tout comme l'algorithme de *Spectral Clustering*, l'algorithme de *Markov Clustering* a l'avantage de comporter essentiellement des étapes de calcul. Cependant, la dernière étape de l'algorithme (consistant à décomposer le graphe en sous-graphes disjoints) peut s'avérer difficile, comme nous l'avions vu en section 2.2 avec l'algorithme WTGPP. De plus, comme le *Spectral Clustering*, celui-ci est coûteux en temps de calcul et en capacité mémoire, avec la manipulation d'une matrice de taille $m \times m$, où m est le nombre de séquences dans le jeu de données.

2.5 Analyse des performances des algorithmes

Pour comparer les différents algorithmes, les chercheurs ont d'abord généré plusieurs jeu de données sur lesquels tester les algorithmes, où chaque jeu de données contient des protéines de proximités différentes. Par ordre de proximité, on a des jeu de données nommés *A-p*, où p signifie que les protéines ont moins de p% de similarité, puis un jeu de données *Gold*, où les protéines sont extrêmement proches.

Ensuite, les chercheurs ont testé les algorithmes avec deux mesures de comparaison : la comparaison séquence-séquence et profil-profil, dont nous avons parlé en section 2.1. Les résultats pour ces deux cas sont présentés en figures 4 et 5. Quatres grandeurs ont été mesurées : la précision, représentant la capacité de l'algorithme à associer une protéine dans la bonne famille, le recall, représentant la capacité de l'algorithme à retrouver une protéine appartenant à une famille, et la F-measure (ou le F-score), qui est une moyenne harmonique entre la précision et le recall. Dans ces tables sont également affichées le nombre de clusters générés.

Table 1 Sequence-sequence comparison F-measure for clustered sequences

	Family															
	TransClust				HiFix				MCL				SCPS			
Dataset	F-measure	Clusters	Precision	Recall	F-measure	Clusters	Precision	Recall	F-measure	Clusters	Precision	Recall	F-measure	Clusters	Precision	Recall
A-10	**0.494**	1757	0.834	0.409	0.467	2780	0.463	0.692	0.352	2310	0.923	0.389	-			
A-20	**0.573**	2013	0.885	0.494	0.491	3270	0.556	0.732	0.398	4125	0.999	0.278	-			
A-30	**0.675**	2561	0.912	0.628	0.583	3749	0.561	0.885	0.415	1827	0.351	0.773	-			
A-50	**0.721**	3221	0.903	0.709	0.608	4861	0.562	0.945	0.457	1912	0.702	0.445	-			
A-70	**0.739**	3486	0.904	0.733	0.630	4921	0.616	0.873	0.474	2323	0.752	0.482	-			
A-90	**0.758**	3630	0.913	0.753	0.653	4973	0.625	0.895	0.511	2824	0.815	0.512	-			
A-95	**0.766**	3715	0.916	0.765	0.654	4992	0.629	0.907	0.527	2873	0.527	0.813	-			
GOLD	**0.914**	96	0.905	0.968	0.902	99	0.960	0.895	0.880	56	0.808	0.942	-			
	Super-family															
A-10	**0.377**	1757	0.917	0.281	**0.337**	2780	0.993	0.274	0.270	3270	0.997	0.180	0.297	658	0.387	0.221
A-20	**0.450**	2013	0.954	0.347	0.362	3270	0.993	0.293	0.282	4024	0.999	0.191	0.352	701	0.400	0.323
A-30	**0.551**	2561	0.551	0.440	0.473	3749	0.994	0.414	0.333	3745	0.998	0.235	0.473	792	0.494	0.364
A-50	**0.609**	3221	0.995	0.499	0.507	4861	0.992	0.457	0.351	3048	0.847	0.310	0.557	753	0.618	0.546
A-70	**0.631**	3486	0.997	0.519	0.539	4921	0.990	0.495	0.377	2086	0.875	0.335	0.581	493	0.649	0.518
A-90	**0.654**	3630	0.996	0.544	0.560	4973	0.989	0.528	0.426	2549	0.922	0.364	0.607	633	0.680	0.531
A-95	**0.659**	3715	0.996	0.552	0.563	4986	0.990	0.542	0.435	2616	0.912	0.378	0.615	940	0.686	0.542
GOLD	0.865	23	1	0.765	**0.915**	13	0.998	0.852	0.827	24	1	0.712	0.904	4	0.864	0.983

Number of clusters found, and weighted mean precision and recall values for each clustering algorithm are shown. Best values are shown in bold.

FIGURE 4 – Performances des algorithmes avec la comparaison séquence-séquence

Table 2 Profile-profile comparison F-measure for clustered sequences

	Family															
	TransClust				HiFix				MCL				SCPS			
Dataset	F-measure	Clusters	Precision	Recall	F-measure	Clusters	Precision	Recall	F-measure	Clusters	Precision	Recall	F-measure	Clusters	Precision	Recall
A-10	**0.741**	1608	0.924	0.732	0.652	2590	0.648	0.916	0.693	783	0.730	0.653	-			
A-20	**0.749**	1773	0.912	0.760	0.685	3022	0.672	0.840	0.703	922	0.736	0.703	-			
A-30	**0.750**	2098	0.868	0.814	0.695	3147	0.678	0.899	0.707	1257	0.731	0.706	-			
A-50	**0.751**	2951	0.860	0.804	0.702	4534	0.702	0.900	0.709	1653	0.724	0.702	-			
A-70	**0.753**	3153	0.858	0.818	0.713	4673	0.709	0.909	0.712	1817	0.727	0.706	-			
A-90	**0.767**	2714	0.833	0.870	0.717	4708	0.889	0.710	0.715	1945	0.743	0.708	-			
A-95	**0.769**	2800	0.766	0.840	0.725	4725	0.709	0.907	0.743	2078	0.768	0.709	-			
GOLD	**0.959**	94	0.950	0.978	0.921	98	0.906	0.918	0.925	81	0.961	0.922	-			
	Super-family															
A-10	0.722	1455	0.997	0.623	0.699	1182	0.963	0.636	**0.752**	714	0.908	0.726	0.750	186	0.742	0.763
A-20	**0.783**	1402	0.990	0.720	0.701	1319	0.964	0.644	0.754	848	0.916	0.738	0.759	253	0.934	0.654
A-30	**0.809**	1676	0.988	0.757	0.705	1500	0.942	0.686	0.778	1062	0.920	0.774	0.777	453	0.914	0.618
A-50	**0.827**	1995	0.987	0.778	0.710	2375	0.964	0.702	0.781	1642	0.968	0.723	0.789	665	0.958	0.693
A-70	**0.833**	2120	0.988	0.783	0.711	2476	0.960	0.707	0.788	1585	0.936	0.782	0.792	758	0.983	0.703
A-90	**0.835**	2213	0.988	0.779	0.715	2524	0.950	0.701	0.805	1799	0.965	0.755	0.805	931	0.993	0.700
A-95	**0.837**	2293	0.989	0.777	0.716	2582	0.960	0.708	0.807	1806	0.948	0.795	0.805	1023	0.995	0.703
GOLD	0.999	6	1	0.999	0.974	7	1	0.953	**1.000**	5	1	1	**1.000**	5	1	1

Number of clusters found, and weighted mean precision and recall values for each clustering algorithm are shown. Best values are shown in bold.

FIGURE 5 – Performances des algorithmes avec la comparaison profil-profil

La tendance générale qui se dégage pour les comparaisons séquence-séquence est que, pour tous les algorithmes, les performances sont moins bonnes lorsque les séquences sont plus éloignées. En revanche, on remarque que, pour la comparaison profil-profil, les performances sont bien meilleures. Enfin, l'algorithme qui semble fournir les meilleurs résultats est l'algorithme de *Transitivity Clustering* avec la comparaison profil-profil [10].

3 Méthodologie

Avant d'expliquer notre méthodologie, rappelons quelle est notre mission dans ce projet. À partir d'un ensemble de gènes[2] $S = \{S_1, S_2, ..., S_m\}$ provenant d'espèces différentes, nous souhaitons, à travers des algorithmes de clustering, répartir ces gènes dans différents groupes d'homologues, ce qui mettra en évidence des ensembles de gènes très similaires et descendant d'un ancêtre commun. Par ce travail, il est possible de reconstruire l'évolution des espèces.

3.1 Simulation de familles de gènes

Pour évaluer efficacement la qualité de nos algorithmes de clustering et vérifier que la répartition des gènes se fait correctement, nous avons besoin de connaître à l'avance les familles auxquelles appartiennent les gènes que nous considérons. C'est pourquoi, comme cela est suggéré dans le sujet, nous avons décidé de créer artificiellement des familles de gènes.

Plus précisément, dans les algorithmes de clustering que nous utiliserons, on part d'un ensemble de séquences, et on génère un arbre répartissant chacune de ces séquences dans des clusters. Pour simuler les familles de gènes, nous allons faire le chemin inverse : nous partons d'un arbre d'évolution des espèces créé artificiellement et d'un ensemble de familles, et nous générons des séquences conformément à cet arbre et à ces familles. Ainsi, nous aurons juste à vérifier que nos algorithmes de clustering retrouvent bien les familles de départ.

Pour ce faire, nous avons utilisé la librairie *AsymmeTree* de *Python* [6]. Tout d'abord, nous avons simulé un arbre phylogénétique entre plusieurs espèces. Nous pouvons spécifier le temps d'évolution à simuler (en millions d'années), le nombre de feuilles dans l'arbre (autrement dit, le nombre d'espèces vivantes à la fin du temps d'évolution), et le modèle d'évolution. Puis, pour chacune des feuilles, nous avons généré le génome de l'espèce associée, en précisant le nombre de familles de gènes à générer. Enfin, nous avons créé des séquences à partir de l'arbre et des génomes. À la fin de la simulation, un dossier "genes/fasta_files" est créé. Dans ce dossier, on a autant de fichiers *fasta* que de feuilles dans l'arbre, et chaque fichier contient un ensemble de gènes pour l'espèce considérée, avec sa famille, comme nous pouvons le voir sur la figure 6.

```
>fam1gene24spec10
GAATAATTATTAACGATACCAGTCTGGCGCATCAATGTCTGAAATTCCACGAATACTAGTCAAGCGTGTGCAAGCGTCAG
ATTAGCCCAACAACACCATACATAACTTTCGGCGCCGCGGTGAGGATGGTCACGCCAAGAAACGGCTGCGTCGACCAAGT
TCAAATACCCGACGTTAAGTCAGAGTAATGTTTTAGCCACCGGGCAACTAA
>fam1gene23spec10
TTTATCGTAGTGTACGGATCAATCGAATTAGTTACACGTCAACGCGAAGACTATGTGACGATACATATTGAAGGACTCTC
AACCCTACATACCCGATCTTGCTACTGTCCATCGACGTCTTTCATTAATGTATGTGCTAAGATGGCACCACGACAAGATG
CAGCACTAGAGCGGCGCATTATAGAATAAATTATACGCTTTCAGCCCACTTAGAAATTTAGATGCTGGAAGTGCCGTGTC
TGGGGATTGCGTGATCGTGATGACGGGGATTGAGTCTATTGAGTCCCA
>fam2gene19spec10
TCTGATTAGATCATACGATATAGATGAACCATGCTTTACTCAGCCTACACAATGGTCGCGGAGGCTGTCCCCTTGGAGCA
AGTGTCCAGGAACATTTTCTTACTTAGATTGGAAGACCAAAGCACCCCAGACTTCAGATGCCAATCCCATCACATGGTCG
ACTAGTGACACCAGCTTTGGATGAGTTGGCGAGGGCAGTC
>fam2gene34spec10
ATGTGGCTGCCTAATCCGCCGAAACACCTGGCGCGTTTTATACACTTGTATACCCAGAACACGAATTGGGGATCCGAGGC
ATAATCATCCCGTATTAGTGCAGTTAACTTACCATTTCTTACATTTCCGAAAGTCAACGGGTGGATGCGATACCAAGCTG
ACTTCCCTTGAACGCTGTG
>fam2gene35spec10
ATAAAAAGCCTCAACCCCTGTTAGGACCGCCCCGCAGCTACGCTTAGCAGACGCTTTCAACGGGTTGAAAGACAGACTAC
GGTCATTTAGAATGGTTAATTGCCCGAAGACAGCGAGGGAATACGTGGCGCAACCAGTCCGGACTATGAATTTCGGGTTG
ACTTGTCTTAAATTTACTCCCAAAAGCCAACGTGTGAATGCAGTGCCAAGCCGCGCGACCCTCGAACAGGTG
>fam2gene26spec10
CCATAAAGCTTCATTGGCTTCATGGAACGCCTTCTTTTACACGTCCGAATTGAGGGAGGTCGTCGAGGAGAAAGGTACCA
CCACTAACCGGAGCTACAATCTGGGGGTTGATCGGATCACATTTACAAAAGCCAACCTGCTGTAGGCCTGGCTATGATGG
GTTCATGTCAGCGACGTG
>fam3gene60spec10
CATTTATTTCCCCGGTTGGTGGTATAAATCCCGACCGAGGGTTGCCGATCTTCCTACCTACCAGGGGGTTGCCATAGATG
CGTGCGATCTCGGAATGGAATGTGCACTC
```

FIGURE 6 – Extrait d'un fichier *fasta* généré par la simulation

En entrée de nos algorithmes, nous fournirons un tableau de séquences *sequences* et un tableau de familles $familles$, où $familles[i]$ est la famille associée à la séquence $sequences[i]$. Pour construire ces tableaux, nous

2. Un gène est un extrait de l'ADN qui caractérise une ou plusieurs propriétés de l'espèce étudiée [8].

avons créé un classe *readSimulateGene*, qui lit chaque fichier *fasta*, extrait les séquences avec leur famille à l'aide des expressions régulières en définissant un *pattern* approprié (figure 7), et stocke ces informations dans les deux tableaux *sequences* et $familles$. En figure 8 sont affichées quelques séquences et familles ainsi simulées.

```
header_pattern = r">fam(?P<family_number>[0-9]+)gene(?P<gene_number>[0-9]+)spec(?P<species_number>[0-9]+)"
```

FIGURE 7 – Pattern utilisé pour identifier la famille du gène

```
Séquences (extrait) :
['TAGGAAACGAGGTGGGGACTTTCGCCATTGGGGACGGGAAGCGTGGACACCCATTGTAAGACAGGGTTCCTGCATTCAGAGAAGGGTGAGGGTCATAGTATTCTCACTAACCGACGTGACCCTTGAGAGGGGACGTTAAAAGCGGACCCCCCCGGCTTGTAGAGACACCCCCGGGTCCAGACTTGCG',
 'TCTTAATAATTACGCGCAACCTAGTTTGTCTCTATTGCCGATCGGTAAACCCACAAGTGACAACAGATGACAGCCTGCTTGTAAAGCCCGTGCCGGACACGCAAAGTGGACTGCCCAGGTCAACAATTAGCATGTTAGTACAGCGTAGATTCCAACAGCGCTATTCAGGAAAAAGGAAAAATTGCCCAA
AGCGGTCAG', 'CCATACGTTCTACTACACGCCATCGCGGTCTCCTGTGTTTTGATGAGAGCTGTCGTTTAGACTGGGTGTCGACTCGGAACTTCGGACAAGACTCCGAACAAGGACTAATGTCCTCGTTTGCTCTGTCTCGGCAACTGCCCTGGAGAAAATTGGATCAGGGCTCATGATTCAATTCATT
CAGATTCCTCCGCTTCAGCTTTTGAC', 'TCATATTTTGACAAGTGAGGTAGTTCCTCCCGTGCTACGTTTGTTCAACTATACCGAGATGGCTATGCAACCAAGAGTGGACCACATGGTGGACGGGTGGGTAATAGGGCCCGCAGGTGCAAGTGGCAGAAACTTGAGCGCAATTATACTCAACAGGGTAA
CCGAGTGCTGGAGTCCGTCCACCGCTCAGCCGTGGTCTAAGCGCAAGGAGCACGGAACGCCTGGA', 'GGGGTTGTAGCAGAGTACCAGAACAACAGGACGCAGTTGCTTCCATGTTACCAAACTGCTCGGGCATTGCAGAGAGCCACATTTATGCTGAAGTGCTTATGTGGGACAGCTCTGGAAGGGAT
TCTGCGGTAAACGTTTAGCCAGGCGGAGCTCGTCTTTTTGAGCCACTGAGGTGCAAGGGCGTCTT']

Familles (extrait) :
['0', '0', '0', '1', '1']
```

FIGURE 8 – Extrait des séquences et familles simulées

3.2 Algorithme des K-Means

3.2.1 Explication de l'algorithme

L'algorithme des K-Means est une méthode permettant de partitionner un ensemble X de données en un certain nombre K de clusters fixé par l'utilisateur. Chaque cluster est représenté par son centre.

On commence par initialiser aléatoirement les centres des K clusters. Puis, on répète les étapes suivantes jusqu'à qu'il n'y ait plus de changement :

1. On assigne chaque point de l'ensemble X au centre le plus proche. On forme alors K clusters.
2. On recalcule les centres de clusters.

Au terme de cet algorithme, on a alors K ensembles disjoints, où chaque ensemble représente un cluster.

3.2.2 Implémentation de l'algorithme

L'algorithme des K-Means présenté ci-dessus fonctionne seulement sur des données géométriques. Or, nous disposons de séquences qui ne sont pas des données géométriques. Pour contrer ce problème, nous pouvons tenter d'appliquer l'algorithme des K-Representatives qui est une variante des K-Means mais pour des données catégoriques. Cependant, cette méthode nécessite de définir la notion de représentant de séquences (l'équivalent des centres). Une solution plus simple que nous avons trouvée est d'appliquer l'algorithme des K-Means sur la matrice de distance des séquences.

Dans une classe *kmeans*, nous avons commencé par générer la matrice de distance. Nous avons calculé cette matrice en nous basant sur la distance de Hamming ou sur la distance de Levenshtein. Ensuite, pour l'algorithme des K-Means, nous avons directement utilisé la librairie *scikit-learn*, qui contient déjà une implémentation de cet algorithme. Nous donnons en entrée de cette méthode notre matrice de distance avec le nombre K de clusters que nous voulons générer, et nous avons en sortie une liste de familles, où le i-ème élément de cette liste correspond à la i-ème séquence du jeu de données initial.

3.3 Algorithme de clustering hiérarchique, méthode agglomérative

3.3.1 Explication de l'algorithme

Le clustering hiérarchique est une méthode qui produit une hiérarchie de partitions d'un ensemble X de données. En sortie d'un algorithme de clustering hiérarchique, on a un arbre, où un cluster est défini comme un sous-arbre de l'arbre initial. La taille du cluster et le nombre de clusters total dépend du niveau

de profondeur auquel on décide de couper la hiérarchie.

Ici, nous utilisons un algorithme de clustering hiérarchique basé sur une méthode agglomérative. Le principe est de partir d'une partition initiale, où on a autant de clusters que de séquences (dans notre cas). Puis, à chaque étape, on cherche le couple (C_i, C_j) qui minimise une mesure de distance. On réitère ce processus jusqu'à l'obtention d'un seul cluster contenant toutes les séquences. Pour obtenir les clusters, on coupe la hiérarchie à un certain niveau de profondeur.

3.3.2 Implémentation de l'algorithme

Le code est très similaire à celui utilisé pour les K-Means. En effet, nous avons, dans une classe *agglomerative*, créé une matrice de distance, soit basée sur la distance de Levenshtein, soit basée sur la distance de Hamming. Puis, nous avons utilisé la classe *AgglomerativeClustering* de la librairie *scikit-learn* pour appliquer l'algorithme. Tout comme pour les K-Means, nous avons en sortie de celui-ci une liste de familles, où le i-ème élément de cette liste corresponc à la i-ème séquence du jeu de données.

3.4 Algorithme UPGMA

3.4.1 Explication de l'algorithme

L'algorithme UPGMA (Unweighted Pair Group Method with Arithmetic Mean) est une méthode qui permet de construire un arbre phylogénétique à partir d'un ensemble de séquences. Cet algorithme est basé sur une matrice de distance D symétrique de taille $m \times m$ au départ, où m est le nombre de séquences. La case $D[i][j]$ correspond à la distance de Levenshtein entre les séquences S_i et S_j. Cet algorithme est donc un exemple de clustering hiérarchique.

On commence par construire la matrice D avec les séquences initiales $\{S_1, S_2, ..., S_m\}$. Puis, tant que D a plus de 1 ligne, on effectue la procédure suivante :

1. On trouve les deux séquences S_i et S_j $(i \neq j)$ telles que $D[i][j]$ soit minimum (on prend les deux séquences les plus proches).
2. On joint ces deux séquences sous un parent commun S_{ij}.
3. On ajoute une ligne et une colonne pour la nouvelle séquence S_{ij} dans D. Pour ce faire, pour chaque colonne k de D, on calcule la moyenne arithmétique de la distance entre S_i et S_k et entre S_j et S_k, autrement dit, on calcule la quantité : $\frac{D[i][k]+D[j][k]}{2}$ pour toute colonne k de D. À noter que la ligne associée à la nouvelle séquence se déduit par symétrie.
4. On supprime les lignes et les colonnes associées aux séquences S_i et S_j. À la fin d'une itération, on a donc une ligne et une colonne en moins dans D.

À la fin de cet algorithme, on a alors un arbre phylogénétique, où les feuilles de cet arbre sont les séquences $S_1, ..., S_m$. Les clusters peuvent alors se déduire visuellement en regroupant les feuilles ayant des parents communs (en coupant à un certain niveau d'antériorité).

3.4.2 Implémentation de l'algorithme

Pour construire l'algorithme UPGMA, nous avons créé une classe *upgma*, et nous nous sommes inspirés d'un exemple d'implémentation déjà existant [4]. Nous n'allons pas parcourir tout le code que nous avons implémenté, mais nous allons détailler quelques éléments clés et les difficultés que nous avons rencontrées.

Pour construire l'arbre, comme cela est fait dans l'exemple [4], nous construisons celui-ci au format *newick*, qui est une représentation sous forme de tuples imbriqués. Par exemple, la représentation (S1,((S2,S3),((S4,S5),S6))) correspond à l'arbre présenté en figure 9.

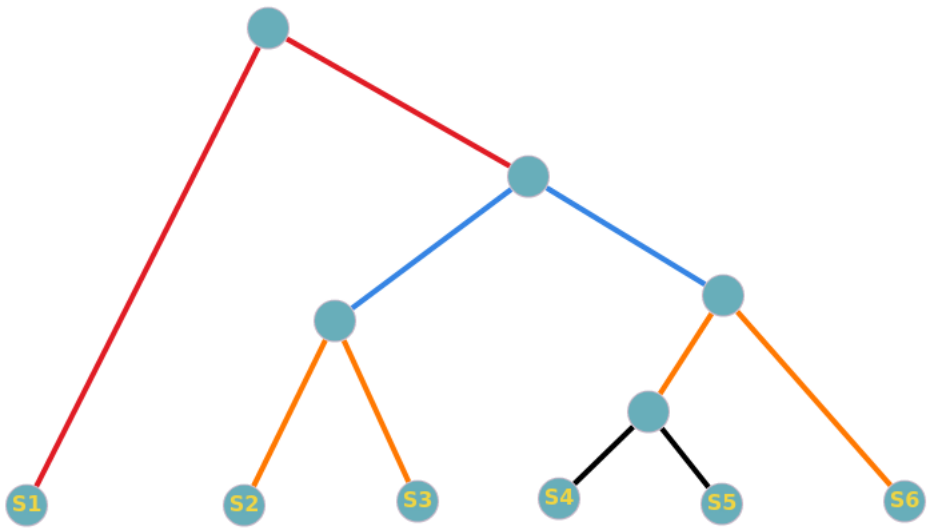


FIGURE 9 – Exemple d'arbre phylogénétique

À partir du tableau de séquences initial (*sequences*), on regroupe itérativement les paires (S_i, S_j) les plus proches en remplaçant $sequences[i]$ par la paire ainsi construite, et en supprimant $sequences[j]$ (pour ne pas avoir deux fois $sequences[j]$). À la fin de l'algorithme, le tableau *sequences* ne contient plus qu'un seul élément, qui est la représentation au format *newick*.

Après avoir généré l'arbre, il faut extraire les clusters. Nous avons eu beaucoup de difficultés à les extraire, et la méthode d'extraction que nous avons retenue n'est clairement pas optimale. Nous avons cherché à procéder de façon récursive. Pour chaque noeud interne, on sépare en 2 familles : la famille du sous-arbre gauche, et la famille du sous-arbre droit. On poursuit ce processus jusqu'à obtenir une feuille, et dans ce cas on ajoute le numéro de la famille associée à la feuille. Le gros problème avec cette méthode, c'est qu'on ne contrôle pas l'extraction des clusters : on ne sait pas combien on va avoir de clusters à la fin, et il se peut aussi qu'une feuille bien isolée à gauche soit associée à la même famille qu'une feuille bien isolée à droite, alors que les deux feuilles ne devraient jamais être associées. Nous avons alors essayé d'implémenter l'extraction des clusters de façon itérative de manière à avoir davantage de contrôle sur le niveau d'antériorité auquel on décide de couper. Malheureusement nous n'avons pas réussi à l'implémenter, et nous avons décidé de passer à d'autres algorithmes de clustering, et de revenir dessus si le temps nous le permet. Nous sommes donc restés sur la méthode récursive.

3.5 Tentative d'implémentation du Transitive Clustering

Nous avons également cherché à implémenter l'algorithme de *Transitive Clustering*, décrit en revue de littérature.

Pour construire la structure de graphe, nous avons utilisé la librairie *networkx*. Si l'initialisation du graphe s'est faite relativement facilement, la transformation de celui-ci en un graphe transitif a été très difficile. Comme nous avons pu le voir en revue de littérature, il s'agit d'un problème d'optimisation sous contraintes, et nous avons pu découvrir dans nos recherches qu'il était possible de résoudre celui-ci en programmation linéaire ([15], section 3.4.4). La librairie *gurobipy* de Python aurait pu nous permettre de le résoudre, mais au vu de la complexité des contraintes, nous avons préféré nous orienter sur l'heuristique présentée en revue de littérature. Malheureusement, celle-ci était très compliquée également, et nous n'avons pas réussi à faire fonctionner l'algorithme présenté. Le fichier *transClust.py* montre les traces de nos recherches.

3.6 Implémentation des mesures d'évaluation des algorithmes

Afin de mesurer les performances des méthodes de classification, et pouvoir les comparer entre elles, nous avons été amené à déterminer une mesure permettant de comparer la liste des familles homologues prédites des séquences par chaque méthode, avec la liste des familles attendues pour ces séquences. Pour cela, nous avons utilisé les métriques d'évaluation standards telles que la justesse, le rappel, la précision et la F-mesure, et ce à partir du calcul de la matrice de confusion. Cette dernière est obtenue en comparant les familles de chaque paire de séquence possible. La comparaison est fait de la sorte suivante :

- Si deux séquences sont dans la même famille et que la prédiction est en accord avec cela, alors on incrémente le nombre de vrai positifs de un.
- Si deux séquences sont dans la même famille et que la prédiction est en désaccord avec cela, alors on incrémente le nombre de faux positifs de un.
- Si deux séquences ne sont pas dans la même famille et que la prédiction est en accord avec cela, alors on incrémente le nombre de vrais négatifs de un.
- Si deux séquences ne sont pas dans la même famille et que la prédiction est en désaccord avec cela, alors on incrémente le nombre de faux négatifs de un.

Ainsi on obtient, pour chaque méthode, une métrique permettant de l'évaluer et la comparer à ses semblables. Il a été nécessaire de calculer la matrice de confusion ainsi, car les étiquettes de familles retournées par les méthodes de classification ne sont pas forcément les mêmes que celles attendues, et par exemple, deux séquences pourraient être dans la même famille, numéroté 0, dans les familles attendues, et seraient dans la même famille également dans les prédiction, mais avec un numéro de famille différent, numéroté 1. Il est donc nécessaire de comparer les séquences une à une afin de savoir quelle est la composition des familles de séquence, et ainsi comparer les résultats attendus aux prédictions.

4 Résultats

Pour évaluer les performances de nos algorithmes, nous nous sommes basés sur quatre métriques, déduites de la matrice de confusion, que nous rappelons ci-dessous :

— L'accuracy, qui représente le taux de bonnes classifications ;
— La précision, qui représente la capacité de l'algorithme à associer un élément à sa bonne classe ;
— Le recall, qui représente la capacité de l'algorithme à retrouver un élément appartenant à une classe ;
— Le F1-score, qui est une moyenne entre la précision et le recall.

Nous avons appliqué ces mesures sur 20 séquences issues de la simulation des gènes.

4.1 Algorithme des K-Means

En utilisant la distance de Hamming pour calculer la matrice de distance, nous obtenons les résultats suivants (figure 10) :

```
Familles prédites :
[6 0 0 6 1 6 1 6 6 1 4 5 5 1 1 3 1 2 6 6]
Familles attendues :
[0, 0, 0, 0, 1, 1, 2, 3, 3, 3, 3, 3, 3, 4, 4, 5, 6, 6, 6, 6]
                      Negative Reality  Positive Reality
Negative Prediction               118                21
Positive Prediction                27                 5
accuracy : 0.7192982456140351
recall : 0.19230769230769232
precision : 0.15625
F1 :  0.1724137931034483
```

FIGURE 10 – Résultats obtenus pour 20 séquences simulées, avec l'algorithme des K-Means et la distance de Hamming

Et avec la distance de Levenshtein, nous obtenons (figure 11) :

```
Familles prédites :
[0 1 1 0 0 0 6 0 0 0 5 3 3 0 0 4 0 2 0 0]
Familles attendues :
[0, 0, 0, 0, 1, 1, 2, 3, 3, 3, 3, 3, 3, 4, 4, 5, 6, 6, 6, 6]
                      Negative Reality  Positive Reality
Negative Prediction                97                17
Positive Prediction                48                 9
accuracy : 0.6198830409356725
recall : 0.34615384615384615
precision : 0.15789473684210525
F1 :  0.21686746987951808
```

FIGURE 11 – Résultats obtenus pour 20 séquences simulées, avec l'algorithme des K-Means et la distance de Levenshtein

Pour les deux distances, on remarque que l'accuracy est plutôt moyenne, avec en moyenne 65% de bonnes classifications. On remarque cependant que l'algorithme a beaucoup de mal à associer un gène dans sa bonne famille, ou à retrouver un gène issu d'une famille, puisque nous obtenons en moyenne environ 15% de précision et 25% de recall.

4.2 Algorithme de clustering hiérarchique (agglomerative clustering)

En utilisant la distance de Hamming pour calculer la matrice de distance, nous obtenons les résultats suivants (figure 12) :

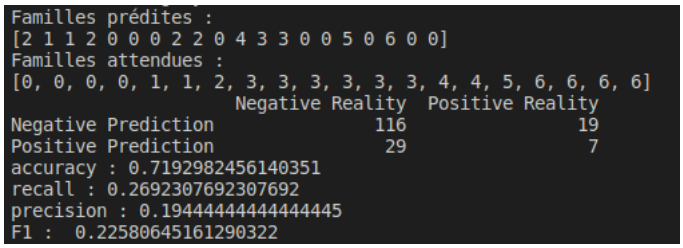


FIGURE 12 – Résultats obtenus pour 20 séquences simulées, avec l'algorithme d'agglomerative clustering et la distance de Hamming

Et avec la distance de Levenshtein, nous obtenons (figure 13) :

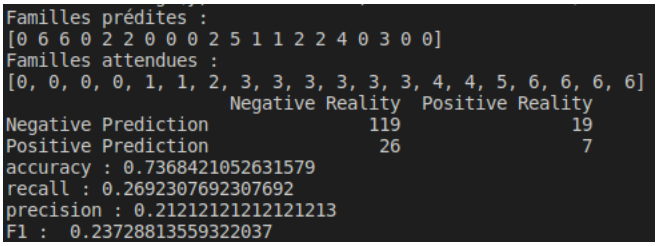


FIGURE 13 – Résultats obtenus pour 20 séquences simulées, avec l'algorithme d'agglomerative clustering et la distance de Levenshtein

On a des résultats qui sont globalement similaires à ceux de l'algorithme des K-Means, bien que les performances soient légèrement meilleures (70% d'accuracy, 20% de précision, et 30% de recall en moyenne).

4.3 Algorithme UPGMA

En utilisant la distance de Hamming pour calculer la matrice de distance, nous obtenons les résultats suivants (figure 14) :

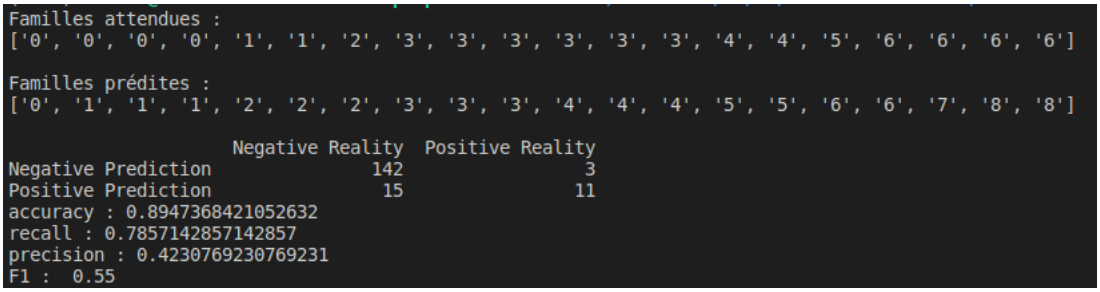


FIGURE 14 – Résultats obtenus pour 20 séquences simulées, avec l'algorithme UPGMA et la distance de Hamming

Et avec la distance de Levenshtein, nous obtenons (figure 15) :

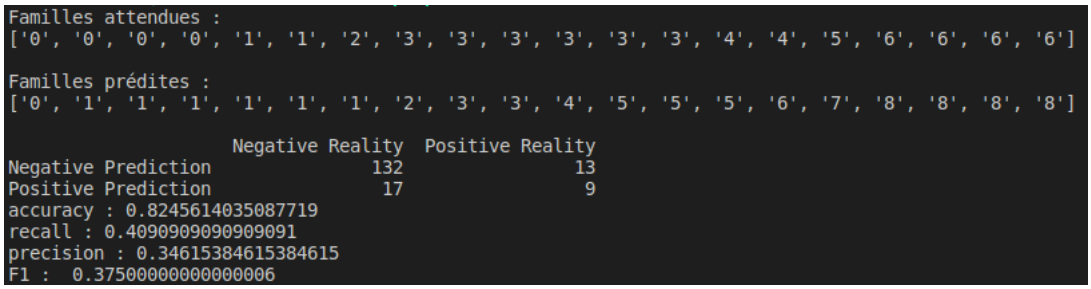


FIGURE 15 – Résultats obtenus pour 20 séquences simulées, avec l'algorithme UPGMA et la distance de Levenshtein

Nous obtenons des résultats un peu meilleurs avec cet algorithme. En effet, l'accuracy est autour des 80%, la précision autour des 40%, et le recall autour des 40% (80% pour la distance de Hamming).

Globalement, on a une accuracy qui est plutôt moyenne à satisfaisante selon l'algorithme considéré. Mais cette mesure n'est pas très pertinente dans notre contexte. En effet, la méthode d'évaluation utilisée favorise grandement les vrais négatifs, puisqu'il y a beaucoup plus de paires de gènes de familles différentes, et il est hautement plus probable que deux gènes ne soient pas de la même famille. Ainsi, le nombre de vrais négatifs augmente et l'accuracy en est améliorée également. Mais, dans notre contexte, nous nous intéressons davantage aux séquences qui sont associées à la bonne famille, et donc à la précision. Cette mesure nous montre que nos algorithmes ont du mal à associer une séquence à sa famille, ce qui laisse présager que nous n'obtiendrons pas de résultats très satisfaisants sur de vrais génomes. Il nous faudrait parfaire les algorithmes que nous avons implémentés pour obtenir de meilleurs résultats. Par exemple, sur l'algorithme UPGMA, nous devrions déterminer les clusters de façon itérative en imposant une profondeur à laquelle on coupe, plutôt que de chercher de façon récursive le plus de clusters possibles.

5 Application sur de vrais génomes

Les méthodes ayant été testées sur des données synthétisées, nous pouvons désormais appliquer la comparaison de génomes à des cas de génomes appartenant à des espèces existantes.

5.1 Jeu de données

Pour cela, nous avons téléchargé, à partir du site de la NCBI [7], des gènes provenant de différentes espèces, ce qui nous permettra d'appliquer nos algorithmes sur des cas réels. Précisons que nous n'avons pas téléchargé les génomes entiers des différentes espèces, car un génome d'une espèce contient ses gènes (qui caractérisent les propriétés de l'espèce [8]), mais également beaucoup de séquences n'apportant aucune information. Concernant le choix des espèces, celui-ci s'est porté sur l'utilisation des gènes de l'homme, du cochon et du moustique, puisque ces deux premières espèces ont des génomes très similaires, par rapport à la majorité des autres espèces, comme le moustique, qui a un génome assez éloigné des deux premières espèces. Notamment, cette similarité génomique est la raison pour laquelle le cochon est souvent utilisé pour modéliser les maladies humaines [5]. Nous nous attendons donc à ce que la comparaison génomique renvoie essentiellement des clusters contenant des gènes des deux espèces. Quant à la comparaison entre le moustique et l'homme, nous nous attendons ici à avoir majoritairement des clusters contenant des gènes d'une seule espèce.

5.2 Calcul de la proximité de génomes

La proximité de génomes est calculée entre deux génomes. Nous calculerons donc à chaque fois la proximité entre les génomes de l'homme et du cochon, et la proximité entre les génomes de l'homme et du moustique.

La proximité entre deux génomes est calculée à partir d'une formule que nous avons mise au point.

On calcule dans un premier temps, pour chaque famille prédite par la méthode de classification utilisée, la proportion de séquences dans la famille par espèce. Par exemple, si la famille 0 comporte respectivement 28 et 15 séquences issues respectivement de l'humain et du cochon, alors la proportion de séquences dans la famille par espèce renvoie respectivement 88% et 12% pour, respectivement, l'homme et le cochon. Dans un second temps, on pondère ces proportions par le nombre total de séquences dans la famille, puis on somme chaque proportion pondérée sur l'ensemble des familles prédites. On divise ensuite la somme par le nombre total de séquences présentes dans les génomes comparés. Le résultat calculé est en fait le calcul de l'écartement des génomes. Ainsi en prenant le complémentaire de l'écartement, on obtient la proximité des génomes.

5.3 Résultats

Nous appliquons la calcul de proximité de génomes aux méthodes de classification décrites dans la section 3. Nous n'utiliserons pas la méthode $UPGMA$ puisque la détermination des clusters de notre implémentation est défaillante. Nous nous concentrons donc seulement sur la méthode des *K-means* et la méthode *agglomérative*. La mesure de proximité entre l'homme et le cochon, et celle entre l'homme et le moustique est présenté en tableau 1.

Méthodes	Homme - Cochon	Homme - Moustique
K-means	65%	76%
Agglomerative	70%	74%

Table 1 – Mesure de proximité inter-espèce pour chaque méthode

Les deux méthodes donnent des résultats assez similaire sur la proximité des espèces. On remarque aussi que selon les résultats de mesure de proximité, l'homme est plus proche du moustique qu'il l'est du cochon. Ces résultats sont donc en opposition avec la littérature. Ceci peut être expliqué par la faible valeur du rappel des méthodes, chaque rappel étant expliqué en section 4, mais aussi par le calcul de proximité qui pourrait être imprécis.

5.4 Ouverture

L'application de la comparaison de génomes à des cas de génomes d'espèces existantes n'ayant pas donné de résultats satisfaisants, nous proposons des pistes d'améliorations. La mesure de proximité est à un stade expérimental, il faudrait la tester et la valider, et changer de mesure dans le cas où elle n'est pas valide. Il serait intéressant de développer plusieurs autres méthodes de clustering pour observer les différences de résultats et aussi obtenir des résultats potentiellement plus exacts. La comparaison de génomes d'espèces réelles a été faites à partir de trois espèces, il serait envisageable de tester les méthodes sur un plus grand nombre d'espèces.

6 Conclusion

Au cours de ce projet, nous avons tenté d'implémenter plusieurs algorithmes de clustering pour regrouper des gènes en familles. En particulier, nous avons appliqué l'algorithme des K-Means, un algorithme de clustering hiérarchique par méthode agglomérative, et l'algorithme UPGMA. Pour évaluer les performances des algorithmes implémentés, nous avons d'abord simulé des familles de gènes de façon à connaître les familles attendues à l'avance. Puis, nous avons construit une matrice de confusion en regardant, pour chaque paire de gènes, si ceux-ci étaient associés à la même famille.

Nous avons pu constater que nos algorithmes ne donnent pas une précision satisfaisante, et que la généralisation sur de vrais gènes ne donnait pas le résultat attendu, à savoir que l'homme est plus proche du cochon que du moustique. Ces mauvais résultats peuvent s'expliquer par des difficultés d'implémentation, notamment dans l'algorithme UPGMA, ainsi que dans notre méthode d'évaluation que nous aurions pu affiner davantage.

Concernant les autres travaux que nous aurions pu faire, nous aurions pu affiner l'extraction des clusters dans l'algorithme UPGMA, implémenter le Transitive Clustering, ainsi que d'autres algorithmes plus efficaces que les K-Means par exemple. Nous aurions aussi pu affiner davantage notre méthode d'évaluation de telle sorte à ce que celle-ci favorise moins les vrais négatifs.

Références

[1] MAS. Saqi A. Paccanaro, JA. Casbon. *Spectral clustering of protein sequences.* 2006.

[2] C.A. Ouzounis A.J.Enright, S. Van Dongen. *An efficient algorithm for large-scale detection of protein families.* 2002.

[3] Wing-Kin Sung algorithms in Bioinformatics (chap 6). *https ://www.comp.nus.edu.sg/ ksung/algo_ in_ bioinfo/l.*

[4] Exemple de code pour l'algorithme UPGMA. *https ://github.com/lex8erna/UPGMApy/blob/master/UPGMA.py.*

[5] Sean J Humphray ; Carol E Scott ; Richard Clark ; Brandy Marron ; Clare Bender ; Nick Camm ; Jayne Davis ; Andrew Jenks ; Angela Noon ; Manish Patel ; Harminder Sehra ; Fengtang Yang ; Margarita B Rogatcheva ; Denis Milan ; Patrick Chardon ; Gary Rohrer ; Dan Nonneman ; Pieter de Jong ; Stacey N Meyers ; Alan Archibald ; Jonathan E Beever ; Lawrence B Schook ; Jane Rogers. *A high utility integrated map of the pig genome.* 2007.

[6] Documentation de la librairie AsymmeTree. *https ://github.com/david-schaller/AsymmeTree/wiki/Manual.*

[7] NCBI Site de téléchargement des gènes. *https ://www.ncbi.nlm.nih.gov/data-hub/genome/.*

[8] Wikipédia définition d'un gène. *https ://fr.wikipedia.org/wiki/Gène.*

[9] NCBI explication de la e value. *https ://www.youtube.com/watch ?v=ZN3RrXAe0uM.*

[10] L. MM Costa G. Zaverucha JS. Bernardes, F. RJ Vieira. *Evaluation and improvements of clustering algorithms for detecting remote homologous protein families.* 2015.

[11] NCBI The Statistics of Sequence Similarity Scores. *https ://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html.*

[12] J. Baumbach M. Martin A. Truss S. Böcker S. Rahmann, T. Wittkop. *Exact and heuristic algorithms for weighted cluster editing.*

[13] A. Paccanaro T. Nepusz, R. Sasidharan. *SCPS : a fast implementation of a spectral method for detecting protein families on a genome-wide scale.* 2010.

[14] A. Truss M. Albrecht S. Böcker J. Baumbach T. Wittkop, D. Emig. *Comprehensive cluster analysis with Transitivity Clustering.* 2011.

[15] Tobias Wittkop. *Clustering Biological Data by Unraveling Hidden Transitive Substructures.* 2010.