**Codon usage**

A00188646 – Alex Savoie

Projet de session pour le cours INFO4301 Apprentissage machine – Automne 2021

Lien vers la présentation : …………………………

**Table des matières**

[**1.** **Introduction** 2](#_Toc88923568)

[**2.** **Base de données** 2](#_Toc88923569)

[**3.** **Algorithmes** 3](#_Toc88923570)

[**3.1** **Régression logistique** 3](#_Toc88923571)

[**3.2** **Random Forest** 4](#_Toc88923572)

[**3.3** **Support Vector Machine (SVC)** 4](#_Toc88923573)

[**3.4** **Neural Network** 5](#_Toc88923574)

[**3.5** **KNN (k-nearest neighbor)** 5](#_Toc88923575)

[**4.** **Optimisation des hyperparamètres** 5](#_Toc88923576)

[**4.1** **Support Vector Machine (SVC)** 5](#_Toc88923577)

[**4.2** **Neural Network** 6](#_Toc88923578)

[**4.3** **KNN (k-nearest neighbor)** 6](#_Toc88923579)

[**5.** **Résultats** 6](#_Toc88923580)

[**6.** **Conclusion** 7](#_Toc88923581)

[**7.** **Références** 7](#_Toc88923582)

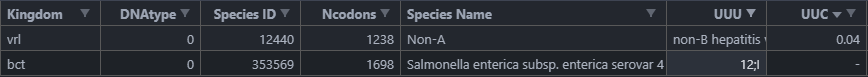
# **Introduction**

Toutes espèces vivantes sur Terre est composé d’un génome distinct. Le génome est composé d’ADN qui sert au développement et la survie de chaque espèce. Dans un long fil d’ADN, une séquence de trois nucléotides d’ADN s’appelle un codon et celui-ci peut être traduit en une acide aminée spécifique. Le but de cette expérience est d’explorer s’il existe un lien entre la fréquence auxquelles un codon se retrouve chez une espèce et le règne de l’espèce. Plus précisément, nous tenterons de développer un modèle qui sera en mesure de prédire de quel règne une espèce fait partie seulement en regardant son ensemble de codons.

# **Base de données**

Notre base de données contient un total de 13028 données et de 69 colonnes. Nos 13028 données correspondent à 13028 différentes espèces vivantes uniques. Notre première colonne, ‘Kingdom’, correspond au règne associé à l’espèce. C’est cette colonne qui sera notre étiquette pour notre classification. Il existe 11 règnes différents possible : 'arc'(archaea), 'bct'(bactérie), 'phg'(bactériophage), 'plm' (plasmide), 'pln' (plante), 'inv' (invertébré), 'vrt' (vertébré), 'mam' (mammifère), 'rod' (rongeurs), 'pri' (primates), et 'vrl'(virus). La deuxième colonnes, nommé DNAType, est la composition génomique. En raison des objectifs de cette expérience, cette colonne sera tout simplement ignoré pour le projet. La colonne SpeciesID est le ID de l’espèce en question. Puisque toutes les données dans SpeciesID sont uniques pour chacune des espèces, cette colonne n’apporte aucun intérêt pour la création de notre algorithme. La prochaine colonne, Ncodons, corresponds aux nombres totaux de codons utilisé qui a servi à calculer la fréquence de chaque codon dans une espèce spécifique. Encore une fois, cette colonne ne présente aucune utilité pour notre projet. La colonne SpeciesName donnent tout simplement le nom de chaque espèce. Pour la même raison que la colonne SpeciesID, elle est ignorée pour ce projet. Les 64 prochaines colonnes sont la fréquence des 64 codons en décimal avec 5 chiffres après la virgule. Les fréquences varient de 0 à 1. Donc, en théorie, la somme de la fréquence des 64 codons devrait être égale à 1. Ce sont ces 64 colonnes qui seront utilisé comme caractéristiques pour nous aider à prédire le règne d’une espèce.

Dans la base de données, il avait deux données corrompues qui causait problème. Puisque notre base de données contient déjà un grand nombre de données, ces deux données ont tout simplement été laissé tomber. Voici ces deux données en question :



Un autre problème que présente notre base de données est l’inégalité de la distribution de nos classes. Certaines classes, comme celles des bactéries et des virus, contiennent des milliers de données alors que des classes, comme celle des plasmides contient seulement une dizaine de données. Cela pourrait causer certains problèmes lors de l’analyse de nos résultats plus tard. Le graphe ci-dessous démontre bien cette inégalité.

Chart, bar chart

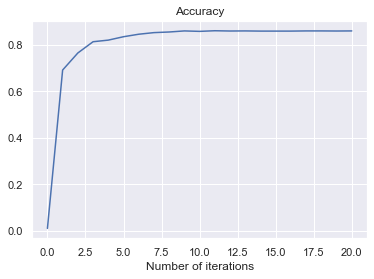
Description automatically generated

Avant d’implémenter les différents algorithmes, il important de mentionner que les données ont été divisé de façon que 80% de celles-ci seront utilisé pour le training et l’autre 20% pour le testing. Les données ont par la suite été standardiser en utilisant le StandardScaler fournit par ScikitLearn , (z = (x-u) / s).

# **Algorithmes**

## **Régression logistique**

Pour débuter, la régression logistique était notre premier algorithme à essayer. Pour sa première exécution, j’ai conservé les paramètres par défaut de ScikitLearn. La régression logistique nous donnait une exactitude d’environ 86%. En observant le graphe ci-dessous, on aperçoit que l’exactitude converge après une dizaine d’itération.



Essayons d’autre algorithme en espérant obtenir des meilleurs résultats.

## **Random Forest**

Le deuxième algorithme implémenté est le classificateur Random Forest. Celui-ci nous amène à un meilleur résultat que la régression logistique. Notre exactitude est augmentée à 89,2% et notre F1-score est passé de 86 à 89%. À noter que, encore une fois, les paramètres par défaut ont été choisi.

## **Support Vector Machine (SVC)**

Encore une fois, notre score à continuer de s’améliorer. Avec le SVC, notre accuracy a augmentée à 93,3%. Il est aussi important à noter, que jusqu’à maintenant, aucun de nos algorithmes a réussi à bien classifier un seul plasmide. Rappelons-nous que les plasmides était l’une des classes les moins fréquentes dans notre base de données. Donc, ce n’est pas vraiment surprenant. Cependant, le SVC à mieux réussi à classifier les classes ayant beaucoup de données comme les bactéries et les vertébrés.

## **Neural Network**

Le réseau neurone démontrait des résultats assez similaires au SVC. Celui-ci avait une exactitude de 92,8%. Dans la catégorie ‘macro avg’, le recall et le f1-score était le meilleur de tous les algorithmes jusqu’à présent. L’une des forces du réseau neurone est qu’il semble être en mesure de mieux classifier les classes ayant peu de données. En plus, le réseau neurone était le premier algorithme à réussir à classifier au moins un plasmide.

## **KNN (k-nearest neighbor)**

Finalement, le dernier algorithme testé fut le KNN. Ce dernier présentait des résultats très similaire au réseau neurone avec une exactitude de 91,9%. Malheureusement, il présentait les mêmes difficultés que le SVC pour classifier les données avec peu de données. Espérons que l’optimisation des hyperparamètres pourra régler ce problème.

# **Optimisation des hyperparamètres**

Pour l’optimisation des hyperparamètres, nous avons choisi trois des cinq algorithmes les plus performant pour les optimisés. Pour compléter cette optimisation, l’outil GridSearchCV a été utilisé. Pour chacun des cas, notre ‘cross-validation’ a été fait avec 5 divisions.

## **Support Vector Machine (SVC)**

Pour le SVC, l’un des hyperparamètres qui à été modifié est le kernel. Par défaut, le SVC utilise le ‘rbf’ mais pour notre cas, nous avons aussi testé avec le ‘linear’. Pour le gamma, nous avons essayer avec ‘scale’ et ‘auto’. Nous avons aussi regardé le comportement du class\_weight en lui donnant une valeur de ‘None’ ou ‘balanced’. Enfin, l’hyperparamètre ‘C’, qui sert à déterminer le taux de régularisation fut essayer avec des valeurs de 0.1, 1 et 10. Notre pourcentage d’exactitude a augmenté de 1,4% (de 93,3% à 94,7%). Voici les meilleurs hyperparamètres :



## **Neural Network**

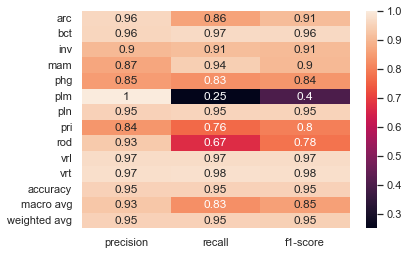
Les hyperparamètres modifié pour le réseau neurone sont : le nombres de couches cachées, le taux d’apprentissage et alpha (paramètre de régularisation L2). Le réseau neurone a le mieux fonctionner lorsqu’il utilise 500 couches cachées (le maximum possible de mes choix). Le meilleur taux d’apprentissage était celui de 0.001 et le meilleur alpha était de 0.01. Notre exactitude à grimper à 94,4% alors qu’il était à 92,8% avec les paramètres par défaut.

## **KNN (k-nearest neighbor)**

Pour le KNN, trois hyperparamètres ont été testés. Effectivement, nous avons observé le comportement du nombre de voisin. Notre algorithme a le mieux fonctionné lorsqu’il utilise seulement 2 voisins. Pour notre algorithme, celui-ci performait mieux avec le calcul de distance de Manhattan plutôt que la distance d’Euclidienne. Finalement, notre modèle fonctionnait mieux lorsque le poids de nos points était de type ‘distance’. Cela signifie seulement que pour chaque point, son poids variait selon sa distance du centre de son voisinage. Grace à cette optimisation des hyperparamètres, notre exactitude a subi une mince augmentation de 92,8% à 93,1%.

# **Résultats**

Notre meilleur modèle était, sans doute, le SVC. À noter que notre SVC fonctionnait mieux lorsque son paramètre de régularisation était élevé (C=10). Lorsque notre valeur de C était de 1, l’exactitude chutait dans les environs de 91%. La différence entre utiliser un gamma de type ‘auto’ versus ‘scale’ ne semblait pas affecter l’exactitude. Pour ce qui est de la kernel utilisé, rbf démontrait de bien meilleur résultat qu’une kernel ‘linear’. Voici le ‘classification report’ contenant des informations pertinente :



Comparé aux autres algorithmes, c’est celui-ci qui a, de manière globale, le mieux performer. Il réussit à bien classifier la grande majorité des classes, surtout celles avec beaucoup de données. Les quatre classes avec le plus de données (bct, vrl, pln et vrt) ont tous de très bon F1-Score. La difficulté est majoritairement rencontrée chez les classes ayant moins de données, comme plm. On remarque que les scores pour les ‘macro avg’ (chaque classe ont le même poids) sont nettement inférieur aux scores des ‘weighted avg’ (le poids des classes en fonction du nombre de données). Cela n’est aucunement surprenant puisque qu’il existe une inégalité au niveau de la distribution des classes. Mais de façon générale, on peut dire que notre algorithme à quand même assez bien performé.

# **Conclusion**

Pour conclure, nous avons réussi à développer un algorithme relativement assez efficace. Pour améliorer le projet, il aurait été pertinent d’avoir plus de données pour les classes qui en avaient peu. Cela aurait fort probablement améliorer nos résultats. Il aurait été aussi intéressant d’essayer un AutoML comme TPOT pour nous aider à trouver un algorithme performant dès le départ. Cependant, puisque notre projet contenait beaucoup de donner, les temps d’exécution pour TPOT était élevé. Toutefois, je suis satisfait des résultats obtenus. Donc on peut dire que les objectifs de ce projet ont été atteints.

# **Références**

1. Tong Yu, Hong Zhu (12th March, 2020) - *Hyper-Parameter Optimization: A Review of Algorithms and Applications (*<https://arxiv.org/abs/2003.05689>)
2. Purva Huilgol (4th September, 2020) - *Precision vs. Recall – An Intuitive Guide for Every Machine Learning Person (*[*https://www.analyticsvidhya.com/blog/2020/09/precision-recall-machine-learning/*](https://www.analyticsvidhya.com/blog/2020/09/precision-recall-machine-learning/)*)*
3. Javaid Nabi (22nd December 2018) – *Machine Learning – Multiclass Classification with Imbalanced Dataset* (<https://towardsdatascience.com/machine-learning-multiclass-classification-with-imbalanced-data-set-29f6a177c1a>)