

UNIVERSITÉ LUMIÈRE LYON 2 - ICOM

PROJET - MASTER 2 DATA MINING

Advanced Supervised Learning

Etudiants :

Mouhamadou Mansour LO

Mame Libasse MBOUP

Encadrant :

Julien AH-PINE

5 avril 2022

Table des matières

1	Introduction	2
2	Sujet à finalité Algorithmique : Random Forest	2
2.1	Thématiques Abordées	2
2.2	Mise en perspective des fondements et Propriétés	3
2.2.1	Les Arbres de Decision - Tree Decision	3
2.2.2	les Forêts Aléatoire - Random Forest	4
3	Algorithmique	4
4	Expérimentations	6
4.1	Description du Jeu de données (Students Performance in Exams)	6
4.2	Résultats expérimentaux et Analyses	7
4.3	Autres méthodes Concurrentes	9
5	Synthèse	11
6	Conclusion	11
7	Liens utiles	11

1 Introduction

Ce projet s'inscrit dans le cadre de notre cours "Advanced Supervised Learning" du Master 2 Data Mining, nous sommes amenés à réaliser des travaux afin d'approfondir nos connaissances théoriques et pratiques des technologies vues en cours dans l'optique de consolider nos compétences. Concrètement, il consiste en l'étude approfondie des arbres de décision et de leur extension par le biais des forêts aléatoires. Le projet est composé de trois parties :

- Implémentation du pseudo-code des arbres décisionnels donné en cours à l'aide de deux fonctions **ArbreGeneration** et **DivisionAttribut** (cas considéré : problème de Catégorisation).
- Implémentation d'une fonction **RandomForest** utilisant les deux fonctions précédentes et mettant en œuvre ré-échantillonnage sur les individus et sur les variables.
- Expérimentations des différentes implémentations sur les jeux de données usuels ainsi que sur un dataset réel et comparaisons des résultats avec les méthodes concurrentes.

Dans ce rapport, nous présenterons d'abord les différentes méthodes vues en cours, ensuite nous détaillerons toutes les procédures d'implémentation de nos modèles. Pour finir, nous ferons une analyse comparative des différents résultats obtenus.

2 Sujet à finalité Algorithmique : Random Forest

2.1 Thématiques Abordées

La problématique de ce sujet repose sur l'implémentation de programmes informatiques qui tentent de résoudre un problème pour lequel nous avons la solution. Globalement, il s'agit de l'apprentissage automatique communément appelé le machine Learning.

Le machine Learning consiste alors à programmer des algorithmes permettant d'apprendre automatiquement de données et d'expérience passées, un algorithme cherchant à résoudre au mieux le problème considéré.

L'un des intérêts du Machine Learning est de construire des modèles avec lesquels on a pas besoin de faire éditer des règles à des experts métier, car justement le but de l'entraînement et de montrer des exemples au modèle afin qu'il décèle de façon automatique des règles qui permettent de classer de nouvelles données. Les avantages sont une très bonne efficacité une économie considérable en terme de main-d'œuvre experte et d'indépendance du domaine. L'apprentissage automatique est principalement classé en trois types : l'apprentissage supervisé, l'apprentissage non supervisé et l'apprentissage par renforcement. Dans notre étude, nous allons nous focaliser sur l'apprentissage supervisé (Supervised Learning) dont le but est d'élaborer un modèle basé sur un jeu de données d'apprentissages et des labels ou étiquettes (nom des catégories ou des classes) et à l'utiliser pour

classer de nouvelles données. Cette technique est utilisée dans plusieurs applications telles que les systèmes de recommandation (youtube, e-commerce), la prédiction des pannes d'équipements, les systèmes de reconnaissance d'images.... D'une manière générale, les algorithmes d'apprentissage supervisé sont utilisés pour résoudre :

- **Classification ou Catégorisation** : les problèmes de prédiction de variable discrète.
- **Régression** : utilisée pour prédire une valeur réelle (variable continue). Par exemple, prédire la température en direct dans une pièce. Quelques algorithmes de classification supervisée : Classification Bayésienne, Machine à vecteurs de support (SVM), Réseau de neurones, Forêts aléatoires, etc.

2.2 Mise en perspective des fondements et Propriétés

Dans le cadre de notre travail, nous comptons mettre en évidence les problématiques théoriques et pratiques de deux méthodes du Machine que sont les arbres de décision et les forêts aléatoires.

2.2.1 Les Arbres de Decision - Tree Decision

Les arbres de décision, également connus sous le nom moderne d'arbres de classification et de régression (CART), ont été introduits par Leo Breiman. Il s'agit d'un algorithme d'apprentissage supervisé qui a une variable cible prédéfinie et qui est surtout utilisé pour la prise de décision non-linéaire avec une surface de décision linéaire simple. En d'autres termes, ils sont adaptables pour résoudre tout type de problème (classification ou régression). Les algorithmes basés sur les arbres sont l'une des meilleures méthodes d'apprentissage supervisé et les plus utilisées. Ils permettent une modélisation prédictive plus précise, plus stable et plus facile à interpréter. Contrairement aux techniques de modélisation linéaire, ils reproduisent assez bien les relations non-linéaires.

Un arbre de décision se compose d'une racine interne qui se divise ensuite en nœuds de décision donc en branches, selon le résultat des branches, la branche suivante ou les feuilles sont formés.

Pour créer un arbre de décision, il suffit de choisir quel attribut doit être testé à chaque nœud de l'arbre. Le gain d'information est la mesure qui sera utilisée pour décider quelle caractéristique doit être testé à chaque nœud. Le gain d'information est lui-même calculé à l'aide d'une mesure appelée entropie, que nous définissons d'abord pour le cas d'un problème de décision binaire, puis pour le cas général. La raison pour laquelle nous avons d'abord défini l'entropie pour un problème de décision binaire est qu'il est plus facile de se faire une idée de ce qu'elle essaie de calculer. Dans les lignes qui suivent nous allons montrer en détails les différentes implémentations des concepts théoriques vus en cours sur les arbres décisionnels. Il s'agira de deux fonctions : **ArbreGeneration** et **DivisionAttribut**

2.2.2 les Forêts Aléatoire - Random Forest

L'algorithme du Random Forest appartient à la famille des méthodes d'ensemble (Ensemble Learning), c'est en fait un cas particulier de bagging (bootstrap aggregating) appliqué aux arbres de décision de type CART. En effet le principe des méthodes de Bagging, et donc en particulier des forêts aléatoires, c'est de faire la moyenne des prévisions de plusieurs modèles indépendants pour réduire la variance et donc l'erreur de prévision. Pour construire ces différents modèles, on sélectionne plusieurs échantillons bootstrap, c'est-à-dire des tirages avec remise. En plus du principe de bagging, les forêts aléatoires ajoutent de l'aléatoire au niveau des variables. Pour chaque arbre, on sélectionne un échantillon bootstrap d'individus et à chaque étape, la construction d'un nœud de l'arbre se fait sur un sous-ensemble de variables tirées aléatoirement. On se retrouve donc avec plusieurs arbres et donc des prédictions différentes pour chaque individu. Dans le cas d'une classification : la catégorie majoritairement choisie par la forêt l'emporte (donc la plus fréquente). En résumé, les forêts aléatoires utilisent à la fois plusieurs ensembles. d'entraînement et plusieurs espaces de description (principe du sous-espace aléatoire ou random subspace). La section suivante présentera l'approche algorithmique décrite en cours par une implémentation d'une fonction **RandomForest** qui utilise les fonctions précédentes et met en œuvre les ré-échantillonnage sur les individus et sur les variables.

3 Algorithmique

La totalité des méthodes algorithmique ont été développées sur Python qui est aujourd'hui, un des outils les plus utilisés dans le domaine de la data science. Ainsi, nous avons travaillé sur l'environnement Spyder(Python 3.9) de la distribution Anaconda. L'idée consiste à implémenter d'abord le pseudo-code des arbres décisionnels en cours (slide 299-300). Premièrement, il nous a été demandé de fournir une fonction `DivisionAttribut` traduisant ce pseudo-code suivant :

```
Fonction : DivisionAttribut  
Input :  $X^m$   
1  $MinE \leftarrow +\infty$   
2 Pour tout Attribut  $X^j$  de  $\{X^1, \dots, X^p\}$  faire  
3   Si  $X^j$  est qualitative avec  $q_j$  catégories faire  
4      $E \leftarrow ent(p^m, X^j)$   
5     Si  $E < MinE$  faire  $MinE \leftarrow E, j^* \leftarrow j$  Fin Si  
6   Sinon ( $X^j$  est quantitative)  
7     Pour toute Séparation en  $\{X^{j,l}, X^{j,r}\}$  possibles faire  
8        $E \leftarrow ent(p^m, \{X^{j,l}, X^{j,r}\})$   
9       Si  $E < MinE$  faire  $MinE \leftarrow E, j^* \leftarrow j$  Fin Si  
10    Fin Pour  
11  Fin Si  
12 Fin Pour  
13 Output :  $j^*$ 
```

FIGURE 1 – DivisionAttribut

Cette fonction permettra de calculer le j^* (une variable) qui minimise l'entropie de l'arbre au nœud m . Le fichier **DivisionsAttribut.py** joint à ce rapport contient le code Python de cette fonctionnalité.

Ensuite il fallait développer la fonction **ArbreGeneration** dont la finalité serait de générer un arbre de décision à partir d'une matrice de données fournie en entrée. Pour cela nous avons créé une classe **Node** (nœud) pour la réalisation de la génération d'arbres. La classe **Node** présente différentes méthodes spécifiques allant de l'initialisation à l'affichage du nœud proprement dit. Les détails de ces deux dernières fonctionnalités sont écrits respectivement dans les fichiers (**ArbreGeneration.py** et **Arbre.py**).

```

Fonction : ArbreGeneration
Input :  $X^m, \theta$ 
1 Si  $ent(p^m) \leq \theta$  faire
2     Créer une feuille et l'étiqueter avec la classe majoritaire
3     Retour
4 Fin Si
5  $j^* \leftarrow \text{DivisionAttribut}(X^m)$ 
6 Initialiser un sous-arbre  $S$ 
7 Pour toute Branche  $m'$  dans  $\{X^{j^*,1}, \dots, X^{j^*,q_j}\}$  faire
8     Déterminer  $X^{m'}$ 
9      $S' \leftarrow \text{ArbreGeneration}(X^{m'}, \theta)$ 
10    Ajouter  $S'$  à une branche de  $S$ 
11 Fin Pour

```

FIGURE 2 – ArbreGeneration

Enfin, les deux fonctions précédentes ont été utilisées pour implémenter l'algorithme du RandomForest vu en cours, donc le but est de mettre en œuvre les ré-échantillonnage sur les individus et sur les variables tel décrit dans le pseudo-code suivant :

```

Input :  $\mathbb{E}, \theta$  (seuil de pureté),  $k$  (nb d'échantillons bootstrap)
1 Pour tout  $j = 1, \dots, k$  faire
2     Déterminer un échantillon bootstrap  $\mathbb{E}^j$ 
3     Induire un ad  $\hat{f}^j$  à partir de  $\mathbb{E}^j$  en appliquant la procédure :
4     Tant que l'arbre n'est pas globalement pur
5         Sélectionner aléatoirement  $r$  attributs
6         Déterminer la meilleure division parmi ces  $r$  variables
7         Séparer le nœud selon la division précédente
8     Fin Tant que
9 Fin Pour
10 Output :  $\{\hat{f}^j\}_{j=1, \dots, k}$ 

```

FIGURE 3 – RandomForest

Le fichier **RandomForest.py** joint a servi au développement de la fonction **RandomForest**. Cette dernière a fait appel à toutes les fonctions décrites plus-haut et a permis de mettre en exergue le principe de la programmation modulaire que nous appliquer. Cet

algo utilise également des fonctions annexes de prédiction qui ont pour rôle de faire des prédictions. D’abord **Makeprediction** qui elle fait appel à **MakeAPrediction**. Enfin, à noter la présence d’une dernière fonction **predictionRandomForest** prenant en entrée (les forêts aléatoires et l’ensemble Test) pour donner en sortie les prédictions majoritaires de chaque forêt.

4 Expérimentations

4.1 Description du Jeu de données (Students Performance in Exams)

Ce jeu de données est constitué des notes obtenues par des étudiants dans différentes matières. Il a été conçu pour comprendre l’influence des parents, de la préparation aux tests, etc, sur les performances des élèves. Il nous vient du site Kaggle. Dans la problématique de notre étude, nous allons essayer de faire des analyses prédictives sur le genre des étudiants en s’appuyant sur notre modèle et à l’aide des différentes caractéristiques de chaque étudiant.

Notre dataset contient 1000 observations et 8 variables qui sont les suivantes :

- gender : le genre male ou female
- race/ethnicity : le groupe ethnique avec modalité A à D
- parental level of education : le niveau d’études des parents
- lunch : le repas du jour
- test préparation course : le degré de révision
- math score : la note en mathématiques sur /100
- reading score : la note en lecture sur /100
- writing score : la note en écriture sur /100

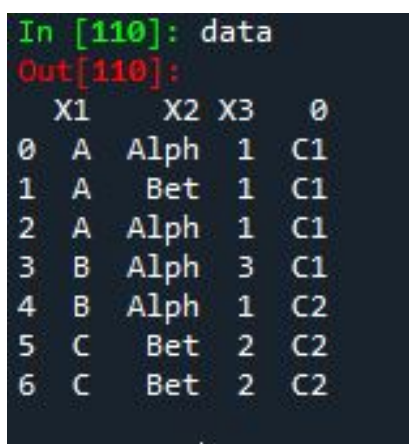
Notre variable cible est “gender”, c’est une variable binaire et prend les modalités : male si l’étudiant est du genre masculin et female s’il est du genre féminin. Nous constatons que le jeu de données est assez équilibré (soit 51.8% contre 48.2%). On note également que le jeu de données est de type mixte avec des variables quantitatives et qualitatives pour profiter de la capacité des arbres à traiter ce type de données. Il présente aussi un nombre important d’individus, donnant ainsi l’occasion de mettre en œuvre les techniques de ré-échantillonnage.

Le fichier **TestStudentPerformance.py** présente l’ensemble des détails du jeu de données et de tous les tests effectués sur nos modes et sur ceux des librairies usuelles. Nous avons joint également un fichier **randomForests sur R** contenant des tests sur le même dataset sur R.

4.2 Résultats expérimentaux et Analyses

Tout d'abord, nous avons appliqué nos algorithmes de Random Forest et de Tree Decision sur les datasets de la librairie Sklearn comme iris, breastcancer et digits afin de comparer les résultats de prédiction des implémentations aux méthodes concurrentes. Voir fichier (`test_Random_Forest.py`).

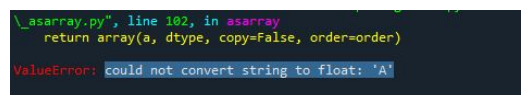
En appliquant `ArbreGeneration` sur un jeu de donnée simulé (voir figure 4), concrètement un jeu de donnée vu en TD, nous remarquons que la librairie Sklearn ne prend pas en entrée des variables catégorielles brutes (renvoie une erreur voir figure 5) et qu'il faut passer par une autre représentation binaire pour ne pas renvoyer d'erreur. On parle alors des techniques de One-hot encoding, que nous utiliserions pour la suite, possible avec la commande `pandas.get_dummies` de la librairie Pandas. Cependant notre modèle `ArbreGeneration` sait très bien prendre en compte une telle forme (catégorielle) de la variable en entrée. Ce qui est déjà un point de bonus pour notre modèle d'arbre de décision.



```
In [110]: data
Out[110]:
```

	X1	X2	X3		
0	A	Alph	1	C1	
1	A	Bet	1	C1	
2	A	Alph	1	C1	
3	B	Alph	3	C1	
4	B	Alph	1	C2	
5	C	Bet	2	C2	
6	C	Bet	2	C2	

FIGURE 4 – Jeu de donnée simulé



```
ValueError: could not convert string to float: 'A'
```

FIGURE 5 – error sklearn

Toujours sur le même fichier nous avons appliqué une subdivision en apprentissage et Test sur tous les datasets utilisés en s'appuyant sur la fonction `train_test_split` en respectant la balance sur les proportions à 33% pour la partie Test afin de pouvoir mettre en oeuvre les procédures communes d'évaluation en Machine Learning. Dans le tableau comparatif ci-après on résume les procédures mise en place avec les modèles d'arbre de décision et Random Forest. On constate que nous avons sur les trois jeux de données utilisés nous obtenons des résultats sensiblement égaux à ceux de Sklearn. Nous avons montré également la supériorité des forêts aléatoires sur les arbres décisionnels. On précise aussi les hyperparamètres pris en compte lors de l'apprentissage, le seuil de pureté pour l'arbre de décision ainsi que le nombre d'arbres pour les forêts aléatoires.

IRIS DATASET :

- ⇒ tree_abre_generation : **0.98** (hyp 0.7)
- ⇒ tree_sklearn : **0.98** (hyp 0.7)

- ⇒ random_forest_model : **0.98** (0.4 ; 200)
- ⇒ random_forest_sklearn : **0.98**

BREAST CANCER :

- ⇒ tree_abre_generation : **0.95744** (hyp 0.28)
- ⇒ tree_sklearn : **0.88** (hyp 0.35)

- ⇒ random_forest_model : **0.95** (hyp 0.4, 500,0.5)
- ⇒ random_forest_sklearn : **0.96** (hyp 200)

DIGITS <=> MNIST :

- ⇒ tree_abre_generation : **0.85** (hyp 0.007)
- ⇒ tree_sklearn : **0.84** (hyp 0.007)

- ⇒ random_forest_model : **0.92** (hyp 0.5, 500,0.1)
- ⇒ random_forest_sklearn : **0.97** (hyp 500)

FIGURE 6 – Tableau comparatif

Pour la partie suivante, toutes les fonctionnalités et méthodes présentées sont décrites dans le fichier **Test_Student_Performance.py**. Concernant notre dataset, nous n'avions pas identifié les quelconques anomalies tant sur les individus que sur les variables pouvant impacter la qualité et la précision des résultats de notre étude. Par contre nous avons converti les variables catégorielles en représentation de vecteurs binaire pour les besoins de conformité de Sklearn tout en sachant que nos modèles savent très bien comment appréhender ces types de variables.

Au regard du taux de reconnaissance (accuracy_score) lors de l'étude sur les arbres de décision, nous pouvons dire que notre modèle est même plus performant que sklearn sur le jeu de donnée considéré. Ce qui veut dire qu'il arrive à mieux détecter le genre de l'étudiant sur l'échantillon Test. On précise que ces résultats reposent particulièrement sur le choix du Thêta. D'où l'importance à trouver les bons hyperparamètres pour la

performance de l'algorithme de prédiction. Dans cette logique, il existe des techniques de sélection automatique et de calibrage du paramètre optimal pour un très grand nombre d'algorithmes statistiques.

```
In [263]: print("SCORE TREE DECISION MODEL:" ,score_tree_model)
SCORE TREE DECISION MODEL: 0.8303030303030303

In [264]: print("SCORE TREE DECISION SKLEARN:" ,score_tree_sklearn)
SCORE TREE DECISION SKLEARN: 0.8181818181818182
```

FIGURE 7 – Accuracy Tree Decision

Par la suite, pour des raisons optimales de temps de calcul, nous avons limité le nombre d'arbres à 100 sur l'analyse du modèle RandomForest. Néanmoins, notre modèle de forêt aléatoire fournit des meilleures prédictions comparées aux arbres de décision. Ce qui traduit exactement ce que nous voulions mettre en évidence dans ce projet et sur des données réelles. Ce qui reste aussi valable pour le modèle de Sklearn.

```
In [273]: print("SCORE RANDOM FOREST MODEL : " , score_random_model)
SCORE RANDOM FOREST MODEL : 0.8575757575757575

In [274]: print("SCORE RANDOM FOREST SKLEARN:", score_random_sklearn)
SCORE RANDOM FOREST SKLEARN: 0.8666666666666667
```

FIGURE 8 – Accuracy Random Forest

4.3 Autres méthodes Concurrentes

Il existe énormément d'autres algorithmes d'apprentissage supervisé permettant de faire de la classification. On peut citer le modèle SVM, qui est considéré comme une généralisation des classifieurs linéaires. On note aussi le gradient qui est une méthode ensembliste comme les random forest, etc. Toutes ces méthodes offrent des résultats aussi satisfaisants que celles vues dans ce rapport. Toujours sur Python, nous avons d'ailleurs testé les réseaux de neurones. Ces derniers ont la réputation de présenter toutefois des résultats impressionnants dans le cadre des problèmes de classification supervisée. C'est sur le même fichier que l'implémentation cette fois-ci sur Keras a été faite :

- d'abord par un perceptron simple avec une couche d'entrée et une couche de sortie, celui-ci produit un assez important score d'accuracy de 0.88
- ensuite un perceptron multicouche ayant 50 neurones dans la couche cachée, et qui logiquement produit le meilleur score parmi tous ces modèles avec une accuracy de 0.89.

```
In [313]: print("SCORE perceptron simple:", score_perceptron)
SCORE perceptron simple: 0.8878787878787879

In [314]: print(PMC.evaluate(X_test,pd.get_dummies(y_test)))
11/11 [=====] - 0s 1ms/step - loss: 0.2443 -
accuracy: 0.8939
[0.24428462982177734, 0.8939393758773804]
```

FIGURE 9 – Accuracy Deep Learning

Sur R, on a la possibilité avec d'autres bibliothèques de faire du machine learning sur les mêmes données. Voyons ce que ça donne. La seule différence dans le prétraitement des données est qu'il faut préciser à R les variables qualitatives en **as.factor**. Les packages `rpart` et `randomforest` permettent respectivement de mettre en pratique l'algorithme tree decision et random forest sur Rstudio. Nous y appliquons les mêmes procédures théoriques : subdivision en échantillons d'apprentissage et de test, apprentissage sur l'échantillon app et prédiction sur le test. Ces travaux sont stockés dans le fichier joint **Random_forest_sur_R**.

```
library(rpart)
arbre <- rpart(gender ~ ., data = app)
print(arbre)

#fonction d'évaluation de la qualité de la prédiction prenant en entrée la variable
#cible observée et la prédiction du modèle
error_rate <- function(yobs,ypred){
  #matrice de confusion
  mc <- table(yobs,ypred)
  #taux d'erreur
  err <- 1.0 - sum(diag(mc))/sum(mc)
  return(err)
}

#prédiction sur échantillon test
pred <- predict(arbre ,newdata=test,type="class")
#taux d'erreur
print(error_rate(test$gender,pred))

#####
#####          RANDOM FOREST          #####
#####

library(randomForest)
#ntree étant le nombre d'arbres
rf <- randomForest(gender ~ ., data = app, ntree = 200)
#prédiction
predrf <- predict(rf,newdata=test,type="class")
#erreur en test
print(error_rate(test$gender,predrf))
```

FIGURE 10 – Pratiques sur R

Globalement, c'est le même constat sur R, à savoir la supériorité des Random Forest sur les arbres. Le taux d'erreur obtenu avec ce dernier est beaucoup plus important que les forêts aléatoires.

```
> print(error_rate(test$gender,pred))
[1] 0.1766667
> print(error_rate(test$gender,predrf))
[1] 0.1533333
> |
```

FIGURE 11 – Pratique sur R

5 Synthèse

Les avantages de nos modèles sont qu'ils proposent des résultats assez satisfaisants et sensiblement égaux à ceux de très grandes bibliothèques utilisées dans la data science comme Sklearn. Les points sur lesquels on note des améliorations sont sur l'optimisation des temps de calcul. Au départ avec en utilisant la bibliothèque pandas notamment dans la fonction `DivisionAttribut` nous obtenions des résultats après des longues minutes d'attente. C'est pourquoi nous avons fait appel aux collections afin d'avoir des résultats moins complexes en temps. Dans l'extension de ces méthodes, il faudrait penser à optimiser les codes dans les différentes fonctions afin d'avoir de meilleurs résultats surtout, lorsque le nombre d'arbres augmente dans les forêts aléatoires.

6 Conclusion

Ces travaux nous ont permis de consolider nos connaissances en programmation en Python et R, en nous familiarisant avec de nombreuses bibliothèques de Machine Learning (Sklearn, Keras, rpart, random forest, etc.). Nous avons eu l'occasion de mettre en pratique les différentes techniques et connaissances acquises durant le cours, notamment, les théories d'arbre de décision et de forêt aléatoires. Hormis les connaissances acquises en programmation Python, ce travail nous a permis aussi de collaborer en équipe qui est un aspect très important dans notre formation éventuellement pour un futur proche, et d'améliorer nos capacités d'analyse. La principale difficulté que nous avons rencontrée dans ce projet a été la démarche générale à adopter pour calibrer les hyper-paramètres des modèles de sorte à trouver les meilleures performances possibles. En effet, comme chaque algorithme, possède ses propres hyper-paramètres, il fallait savoir exactement les bons attributs de la bibliothèque à mettre pour les estimer. Ce qui nécessitait de lire la documentation de chaque bibliothèque pour pouvoir faire la correspondance entre les hyper-paramètres des algorithmes utilisés et ceux proposés par ces dernières.

7 Liens utiles

<https://www.lovelyanalytics.com/2016/08/20/random-forest-comment-ca-marche/>
<https://towardsdatascience.com/decision-trees-d07e0f420175>