UNIVERSITÉ DE SHERBROOKE

Projet Techniques d'Apprentissage

Leaf Classification

Lilian FAVRE GARCIA 21 153 421 favl2301@usherbrooke.ca

Tsiory RAZAFINDRAMISA 21 145 627 raza3902@usherbrooke.ca

Sommaire

I / Contexte du projet

Dans ce projet, nous avons choisi de baser notre thème sur la classification des feuilles proposé par le sujet (lien vers kaggle). Nous allons ainsi essayer de classifier des feuilles d'arbres selon leur espèce. Les modèles que nous avons utilisés sont tous issus de la librairie *Sklearn*.

Dans ce projet, nous avons commencé par traiter les données fournies, puis nous les avons envoyées à nos modèles pour soit réaliser une k-cross validation et entrainer nos modèles sur les meilleures valeurs retenue, soit entrainer le modèle avec les valeurs que nous avons implémentées par défaut que nous avons trouvé lors de nos tests.

Notre projet est disponible en ligne sur notre GitHub ici.

Pour lancer le projet en ligne de commande, il est nécessaire d'indiquer les paramètres suivants :

Exemples :

- « *python main.py Data 0 1 -1 1 5 knn 1/2/3 10/50 0* »
- « *python main.py Data 0 1 3 1 5 svc 1/2/3 rbf/poly scale/auto* »
- « *python main.py Data 0 1 -1 0 nusvc* »

- Arguments obligatoires, suivit des arguments des modèles ci-dessous :
    - Le chemin du dossier Data contenant tous les fichiers du projet
    - Un booléen (0/1) indiquant si vous souhaitez afficher les données sous forme d'histogrammes
    - La proportion]0.0, 1.0] maximum des données d'un caractéristique que l'on accepte dans un intervalle de 10% des données totales ce cette même caractéristique (des explications plus détaillées sont dans le rapport). Nous avons indiqué dans notre rapport que ce paramètre n'est pas pertinent mais nous l'avons laissé au cas où. Pour sélectionner toutes les données ce que nous recommandons au vu des résultats obtenus, mettre 1.
    - Le nombre de compositions [3-10] Test/Valid possibles faisable pour limiter la « chance des bons résultats ». Le dataset Train sera séparé en ce nombre de sous-dataset, puis chacun d'entre eux sera tour à tour le dataset Validation (des explications plus détaillées figurent dans le rapport). Mettre -1 pour ne pas créer de sous-dataset.
    - Un booléen (0/1) pour autoriser ou non la recherche d'hyperparamètre
        - Si oui, indiquer ensuite le nombre de K-cross validation. Pour choisir des nombres possibles pour réaliser la re cherche d'hyperparamètre, il suffit d'en écrire plusieurs en les séparant par des « / ». Par exemple, pour le modèle KNN, on peut écrire.

Si vous choisissez le modèle KNN :

- Noter « knn ».
- Si vous ne mentionnez rien ensuite, le modèle prendra les valeurs par défaut que nous avons obtenues lors de nos tests dans les recherches d'hyperparamètres.
- Sinon, il faut impérativement inscrire les arguments suivants :
    - Le nombre indiquant le nombre de K voisins sélectionnés [1,192[
    - Le nombre indiquant la taille de la feuille [1,192[
    - Un booléen indiquant s'il faut utiliser la mesure de la distance euclidienne (0) ou alors la mesure de Manhattan (1)

Si vous choisissez le modèle Random Forest :

- Noter « rforest ».
- Si vous ne mentionnez rien ensuite, le modèle prendra les valeurs par défaut que nous avons obtenues lors de nos tests dans les recherches d'hyperparamètres.
- Sinon, il faut impérativement inscrire les arguments suivants :
    - Le nombre d'arbres qui seront utilisés dans la forêt pour calculer le résultat final [10, 1000]
    - La profondeur maximale d'un arbre [1, 192[
    - Le nombre minimum qu'il faut pour séparer une node en feuilles [1,64[
    - Le nombre indiquant le nombre de caractéristiques qu'il faut trouver lors d'une séparation [1,64[
    - Un string indiquant quelle mesure de la qualité d'une node il faut utiliser {« gini » ou « entropy »}

Si vous choisissez le modèle Gradient Boosting :

- Noter « gboost ».
- Si vous ne mentionnez rien ensuite, le modèle prendra les valeurs par défaut que nous avons obtenues lors de nos tests dans les recherches d'hyperparamètres.
- Sinon, il faut impérativement inscrire les arguments suivants :
    - Le nombre indiquant le taux d'apprentissage [0.01, 1]
    - Le nombre d'étapes de boosting à réaliser [10,1000]
    - Le nombre minimum qu'il faut pour séparer une node en feuilles [1,64[

Si vous choisissez le modèle Linear Discriminant Analysis :

- Noter « lda ».
- Si vous ne mentionnez rien ensuite, le modèle prendra les valeurs par défaut que nous avons obtenues dans les recherches d'hyperparamètres.
- Sinon, il faut impérativement inscrire les arguments suivants :
    - Le solveur à utiliser ['svd', 'lsqr', 'eigen']
    - Le 'shrinkage' qui peut être None, automatique ('auto') en utilisant le lemme de Ledoit-Wolf ou un float entre 0 et 1, il ne fonctionne que si le solveur est 'lsqr' ou 'eigen'

Si vous choisissez le modèle NuSVC :

- Noter « nusvc ».
- Si vous ne mentionnez rien ensuite, le modèle prendra les valeurs par défaut que nous avons obtenues dans les recherches d'hyperparamètres.
- Sinon, il faut impérativement inscrire les arguments suivants :
    - Le type de noyau à utiliser dans l'algorithme ['rbf', 'linear', 'poly', 'sigmoid'].
    - Le coefficient du noyau gamma ['scale', 'auto'] si le noyau est 'rbf', 'poly' ou 'sigmoid'. Gamma='scale' signifie que la valeur de gamma est 1 / (n_features * X.var()), et gamma='auto' signifie 1 / n_features.

Si vous choisissez le modèle SVC :

- Noter « svc »
- Si vous ne mentionnez rien ensuite, le modèle prendra les valeurs par défaut que nous avons obtenues dans les recherches d'hyperparamètres.
- Sinon, il faut impérativement inscrire les arguments suivants :
    - Le paramètre de régularisation C, strictement positive [1, 10, 100, 1000, 10 000, 100 000]
    - Le type de noyau à utiliser dans l'algorithme ['rbf', 'linear', 'poly', 'sigmoid'].
    - Le coefficient du noyau gamma ['scale', 'auto'] si le noyau est 'rbf', 'poly' ou 'sigmoid'. Gamma='scale' signifie que la valeur de gamma est 1 / (n_features * X.var()), et gamma='auto' signifie 1 / n_features.

II / Nos données

Le set de donnée fourni contient deux sous-sets d'informations sur des feuilles. Le premier étant une banque de 1584 images et le second, trois fichiers .csv contenant des informations sur chacune des feuilles. Dans les tableaux .csv figure un fichier d'entraînement ainsi qu'un fichier de test. Les fichiers d'entraînement et de test sont les mêmes, à la différence près que le fichier de test ne possède pas le nom de l'espèce à laquelle elle appartient, plus précisément la classe de sortie. Ce qui ne permet pas réellement de tester la véracité les données sorties.

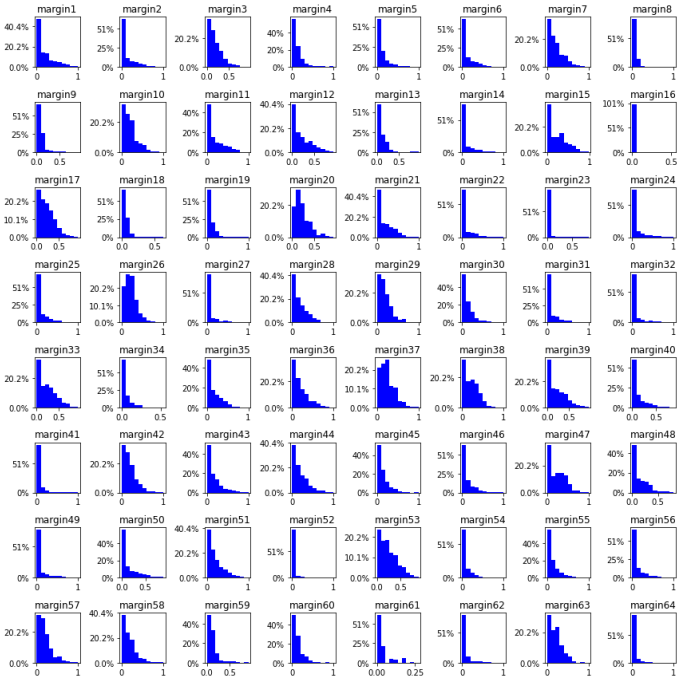
Dans les fichiers .csv, les données sont, en plus du nom des feuilles et de leur ID, composées de 64 caractéristiques de 3 types de caractéristiques différentes. En effet, chaque feuille possède 64 valeurs de « marge », 64 valeurs de « texture » ainsi que 64 valeurs de « forme » ou shape. Nous nous retrouvons donc avec 192 valeurs différentes à traiter.

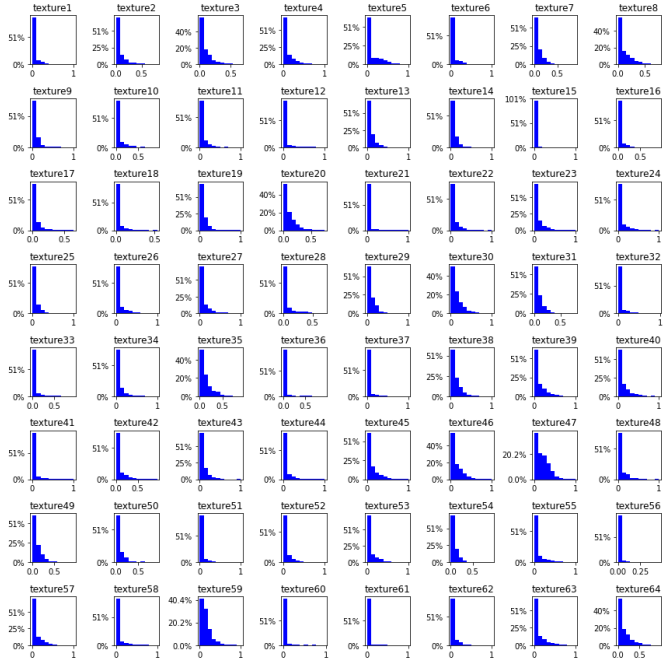
III / Le traitement des données

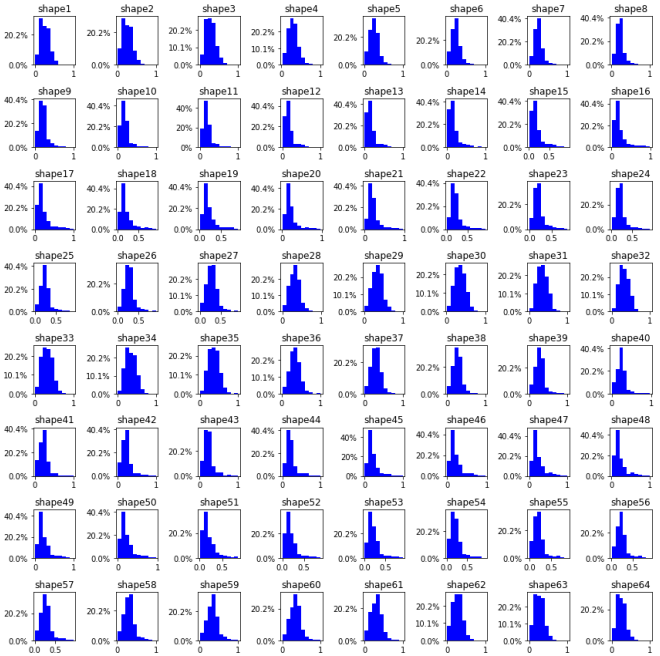
Dans tout projet de science des données, la partie traitement des données est sûrement la plus importante. En plus de rendre les données traitables, cette partie doit faire en sorte de garder le plus d'informations tout en ne biaisant pas ces dernières. Aussi, afin d'améliorer la capacité de nos algorithmes, nous avons développé une partie traitement des données. Tout ce traitement s'effectue dans le fichier *Dataset.py* et reste le même pour tous les algorithmes. Ce pré-traitement s'effectue en deux parties. La première étant automatique, la seconde est un choix de variable selon l'utilisateur.

Tout d'abord, nous avons effectué une recherche automatique des données dans les .csv pour savoir s'il manquait des données ou bien si certaines d'entre elles étaient corrompues. Comme les données fournies étaient complètes, nous n'avons pas eu besoin d'ajouter des étapes de pré-traitement.

Ensuite, pour choisir les traitements à réaliser, nous avons tout d'abord affiché les données brutes sous forme d'histogrammes pour avoir une représentation visuelle. Nous remarquons que la marge et les textures ont une allure inversement proportionnelle contrairement à la forme générale des feuilles qui ont des courbes à l'allure gaussienne (ce sont ici les données présentes dans le fichier train).


margin1
margin2
margin3
margin4
margin5
margin6
margin7
margin8
margin9
margin10
margin11
margin12
margin13
margin14
margin15
margin16
margin17
margin18
margin19
margin20
margin21
margin22
margin23
margin24
margin25
margin26
margin27
margin28
margin29
margin30
margin31
margin32
margin33
margin34
margin35
margin36
margin37
margin38
margin39
margin40
margin41
margin42
margin43
margin44
margin45
margin46
margin47
margin48
margin49
margin50
margin51
margin52
margin53
margin54
margin55
margin56
margin57
margin58
margin59
margin60
margin61
margin62
margin63
margin64


texture1
texture2
texture3
texture4
texture5
texture6
texture7
texture8
texture9
texture10
texture11
texture12
texture13
texture14
texture15
texture16
texture17
texture18
texture19
texture20
texture21
texture22
texture23
texture24
texture25
texture26
texture27
texture28
texture29
texture30
texture31
texture32
texture33
texture34
texture35
texture36
texture37
texture38
texture39
texture40
texture41
texture42
texture43
texture44
texture45
texture46
texture47
texture48
texture49
texture50
texture51
texture52
texture53
texture54
texture55
texture56
texture57
texture58
texture59
texture60
texture61
texture62
texture63
texture64


shape1
shape2
shape3
shape4
shape5
shape6
shape7
shape8
shape9
shape10
shape11
shape12
shape13
shape14
shape15
shape16
shape17
shape18
shape19
shape20
shape21
shape22
shape23
shape24
shape25
shape26
shape27
shape28
shape29
shape30
shape31
shape32
shape33
shape34
shape35
shape36
shape37
shape38
shape39
shape40
shape41
shape42
shape43
shape44
shape45
shape46
shape47
shape48
shape49
shape50
shape51
shape52
shape53
shape54
shape55
shape56
shape57
shape58
shape59
shape60
shape61
shape62
shape63
shape64

Concernant le traitement des données, nous avons essayé 3 traitements, mais n'avons gardé que les deux premiers car la troncation n'était pas pertinente lors des tests.

1. Centre - réduction

Une partie de nos données étant gaussiennes, nous effectuons un centrage-réduction des données afin que les algorithmes discernent mieux les valeurs en-dehors de la moyenne, et que les comparaisons de valeurs se fassent plus facilement grâce à une dispersion des valeurs équilibrées.

2. Normalisation

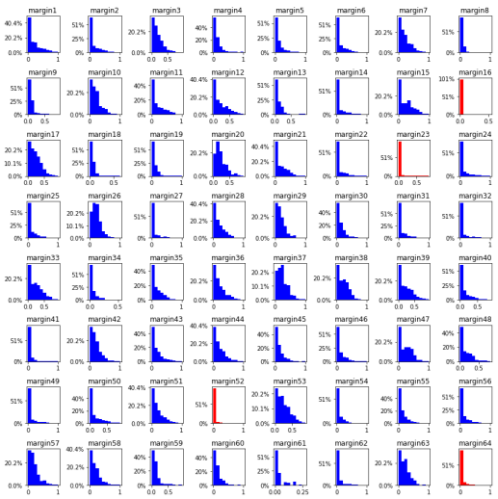
Cette étape permet de remettre les valeurs qui auraient été négatives lors de l'étape précédente entre 0 et 1. Celle-ci est importante car sans cette étape, les variables n'auraient pas réellement de sens. De plus, elle permet de comparer les données entres elles en les égalisant sur une échelle commune.

3. Troncation

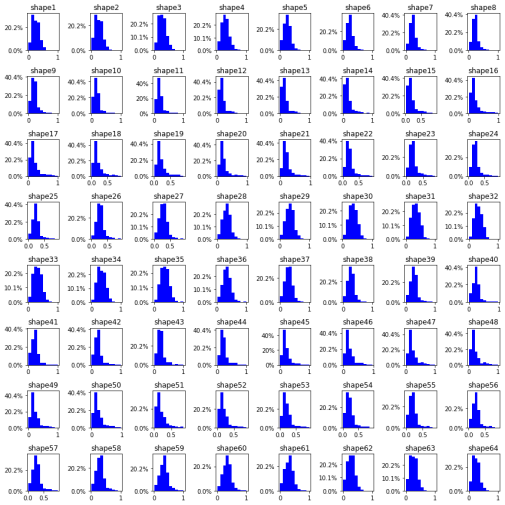
L'objectif théorique étant d'avoir le plus d'informations possible entre les données dans le but de les séparer. Nous avons pourtant choisi de tronquer les valeurs des données lors de l'algorithme du KNN ainsi que les autres afin de ne pas garder trop de données, et pour pouvoir obtenir plus de correspondances. Cependant, les résultats n'étaient pas meilleurs, voire pire. Nous avons, par conséquent, abandonné cette étape du traitement des données.

Enfin, nous avons ajouté une dernière étape de sélection des données. En effet, le but étant de réussir à séparer les données. Nous avons alors implémenté une fonction *feature_selection(N)* qui permet de choisir les données souhaitées pour les algorithmes. Cette fonction prend en paramètre un nombre N entre 0 et 1 choisi par l'utilisateur. Cette fonction va prendre les histogrammes tels que ceux présentés plus haut, et va choisir quelles données enlever à partir de la variable N. N représente ainsi la proportion maximale de données que l'on accepte dans un intervalle de 10% entre la valeur minimale et la valeur maximale d'un histogramme. Autrement dit, les histogrammes étant composés techniquement de 10 bins, le programme ne sélectionnera que ceux qui ont moins de N% de leurs données dans leur bin le plus grand. Par exemple, si l'utilisateur rentre une valeur de 0.9, l'histogramme de texture 60 ne sera pas choisi. On peut, en effet, aisément voir que le bin le plus gros possède plus de 90% de toutes les valeurs.

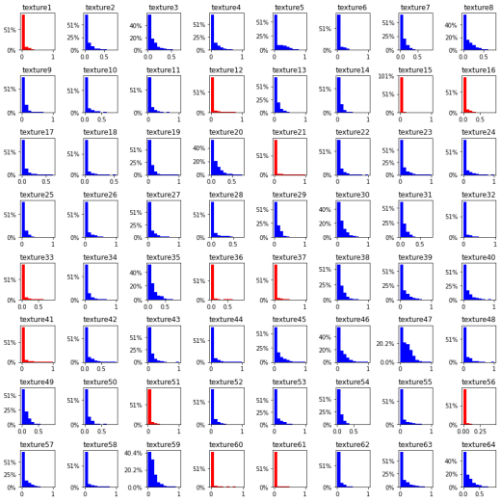
Cette sélection permet donc de réduire la quantité de données traités en éliminant les caractéristiques qui ont très peu de variations et qui, par conséquent, ne permettent pas réellement de différencier les données. En sélectionnant un N = 0.85, on se retrouve, par exemple, avec cette sélection (les histogrammes en rouges sont ceux qui seront supprimés après-coup dans les sets de données Train et Test) :



Histogramme des marges avec N=0.85



Histogramme des formes avec N=0.85



Histogramme des textures avec N=0.85

Nous pensions que cette sélection serait pertinente si nous considérons un ordinateur avec peu de puissance de calcul et si nous ne souhaitons sélectionner qu'une partie des données. Il devait s'agir d'un compromis entre quantité de données traités et temps de calcul. Cependant, cette hypothèse n'a pas été vérifiée lors de nos tests : le temps de calcul n'a quasiment pas changé sur les plus longs (12h). Nous avons par conséquent laissé ce paramètre, mais nous recommandons de le mettre tout le temps à 1 afin de sélectionner toutes les données.

Pour récapituler cette partie de traitement des données, nous pouvons remarquer qu'elle est assez courte. Les données de base fournies par Kaggle est, en effet, extrêmement bien fait, sans erreurs de données manquantes ou autre données très aberrantes. Ce qui, de manière générale, nous semble suffisamment rare dans les différents dataset auxquels nous avons eu affaire, pour avoir besoin de signifier la qualité de celui-ci. Nous n'avons alors pas eu besoin de pousser plus loin le traitement.

IV / Modèles et recherche d'hyperparamètres

Le set de données initial comporte deux fichiers Train avec les labels et Test sans. Nous n'allons pas utiliser ce fichier de set Test pour les essais suivants. Cependant, ce fichier comportant le nom Test peut porter à confusion comme il ne peut pas être utilisé pour les tests. Celui-ci peut être renommé « Inconnu » par exemple. Ainsi, nous créons un 3ème set de donnée test « Test » crée à partir du fichier Train qui comporte 20% des données de Train lors du fonctionnement du programme.

Tous les modèles que nous avons intégrés dans notre projet sont ceux implémentés par *Sklearn*.

Aussi, afin d'appliquer la méthode de la cross-validation pour chaque algorithme, nous avons utilisé *train_test_split* de *Sklearn* dans le but de subdiviser nos données d'entrainement Train en données de Validation avec également un ratio de 80%-20%. Afin de tester plusieurs valeurs d'hyperparamètres possibles, nous avons bouclé notre code sur des listes de valeurs de chaque paramètre possible. Cette

recherche d'hyperparamètre n'est donc pas aléatoire. Nous avons essayé plusieurs ordres de grandeurs pour chaque valeur, ensuite nous avons mis par défaut les meilleures valeurs sorties lors de nos tests dans l'appel des fonctions de la construction de nos modèles. Mais l'utilisateur pourra toujours spécifier des valeurs particulières le cas échéant.

Enfin, après l'entraînement de chaque modèle avec chaque composition d'hyperparamètre, nous testons ce modèle sur le set de donnée Validation. C'est donc sur ce dernier set qu'on enregistre le résultat moyen des K essais afin d'observer la capacité des paramètres testés (courbes en orange sur les captures d'écran suivants).

Toutes ces recherches ont, comme nous l'avons recommandé plus haut, été effectuées avec le paramètre *feature_selection(N)*, N=1,0.

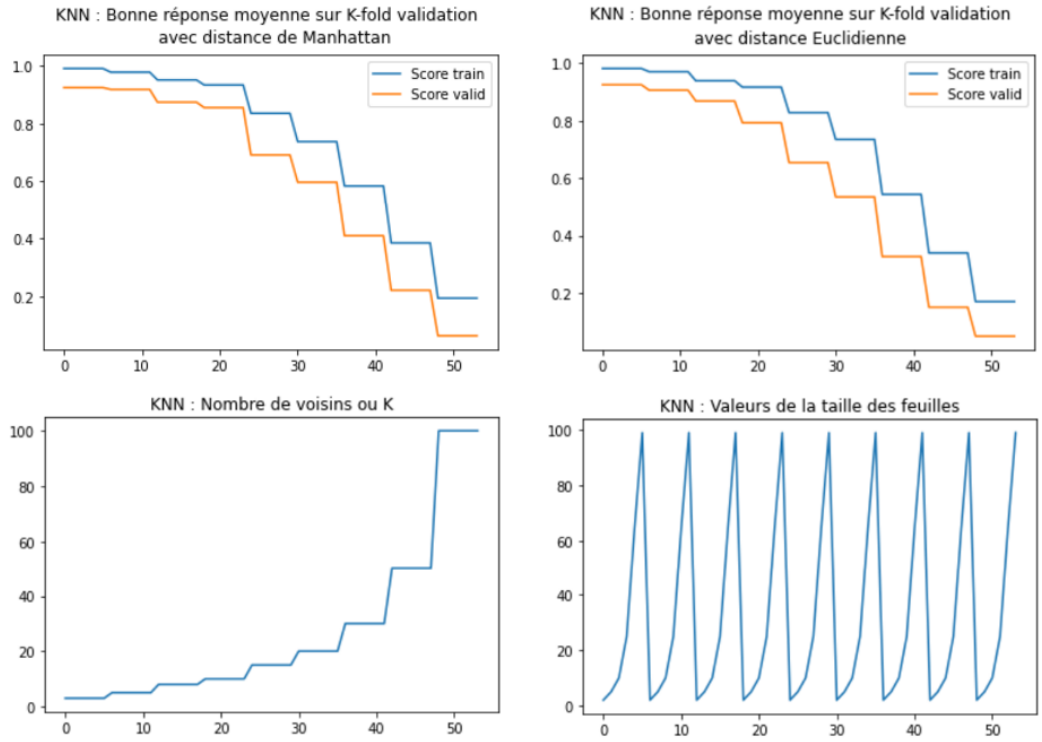
1. K-Nearest Neighbour

Ce modèle est le plus simple parmi tous les modèles d'apprentissage supervisés. En effet, pour choisir une classe de sortie, le modèle va retenir les K plus proches voisins en termes de distance dans l'espace des données et sélectionner la classe majoritaire.

Pour cette recherche d'hyperparamètres, nous avons testé les valeurs suivantes :

Nombre de K voisins : 3, 5, 8, 10, 15, 20, 30, 50, 100

Nombre de feuilles : 2, 5, 10, 25, 64, 99



Kcross Validation du modèle KNN

Nous avons donc séléctionné les valeurs suivantes. Nous les avons implémentées comme valeurs par défaut :

K-voisins = 3 *nb_feuille* = 2 *Manhattan_distance = True*

Sur ces résultats, nous pouvons facilement observer que moins il y a de voisins pris en compte, plus le résultat est bon. Puis contrairement à ce que nous nous attendions, il y a moins d'overfitting avec un nombre de voisins bas, comparé aux valeurs de la courbe plus à droite (l'axe des X > 30), qui ont un écart de valeurs entre le Train et le Validation plus important. De plus, les mesures de distance utilisées (Manhattan et Euclidienne), sont quasiment similaires, avec néanmoins de meilleurs résultats pour la distance de Manhattan.

2. Random Forest

Ce modèle a pour but de réduire la variance d'un arbre de décision. Pour cela, le modèle utilise plusieurs arbres de décision avec la technique de bagging. Ce modèle en particulier est plutôt précis lorsque le problème possède de nombreuses classes. De plus, il est résistant aux valeurs aberrantes.

Pour cette recherche d'hyperparamètres, nous avons testé les valeurs suivantes :

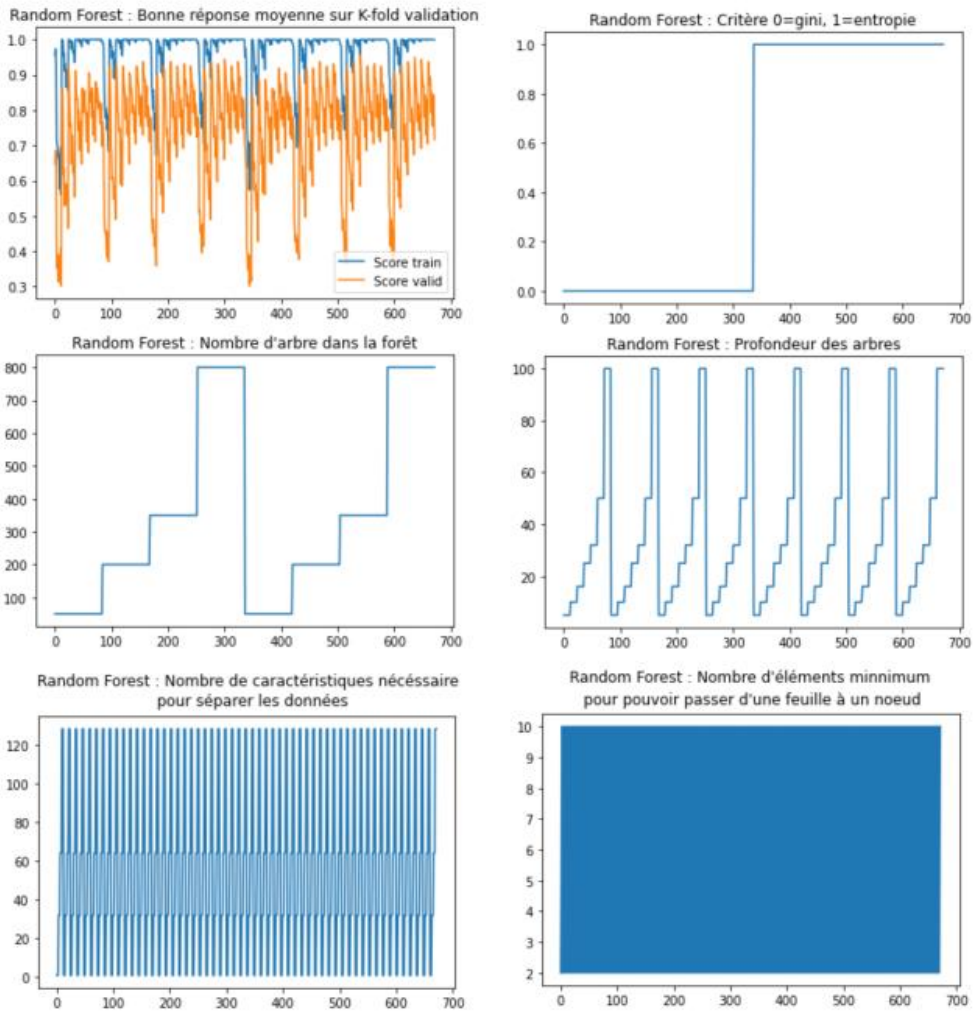
Nombre d'arbres : 50, 200, 350, 800

Profondeur maximale : 5,10,25, 50, 100

Critère d'évaluation d'une node : Gini, Entropie

Nombre de caractéristiques pour créer une nouvelle node : 1, 32, 64, 128

Nombre de caractéristiques minimum pour séparer les données : 2,5,10



Ce programme a donc séléctionné les valeurs suivantes que nous avons implémentées comme valeurs par défaut : (800 arbres a été la meilleure valeure mais nous expliquons notre choix dans la partie VI). Ces paramètres sont ceux qui ont obtenu le meilleur score avec les données Validation et non Train.

*Nb*_trees = 350 max_*feature* = 1 *depth* = 50 *criterion* = *entropy* min_*sample* = 2

Pour ce second modèle, nous remarquons que le paramètre le plus important est la profondeur des arbres qui peut changer de presque 50% la qualité du score obtenu. Il est, en effet, logique que la profondeur des arbres permette de trier plus facilement des éléments. Le second paramètre qui influe est le nombre d'arbres dans la forêt, qui permet donc de moyenner encore plus les résultats et ainsi, d'être plus précis sur le résultat final. Néanmoins, cela entraîne un temps de calcul plus long. Enfin, un dernier paramètre qui importe grandement est le nombre de caractéristiques nécéssaires pour séparer les données

3. Gradient Boosting

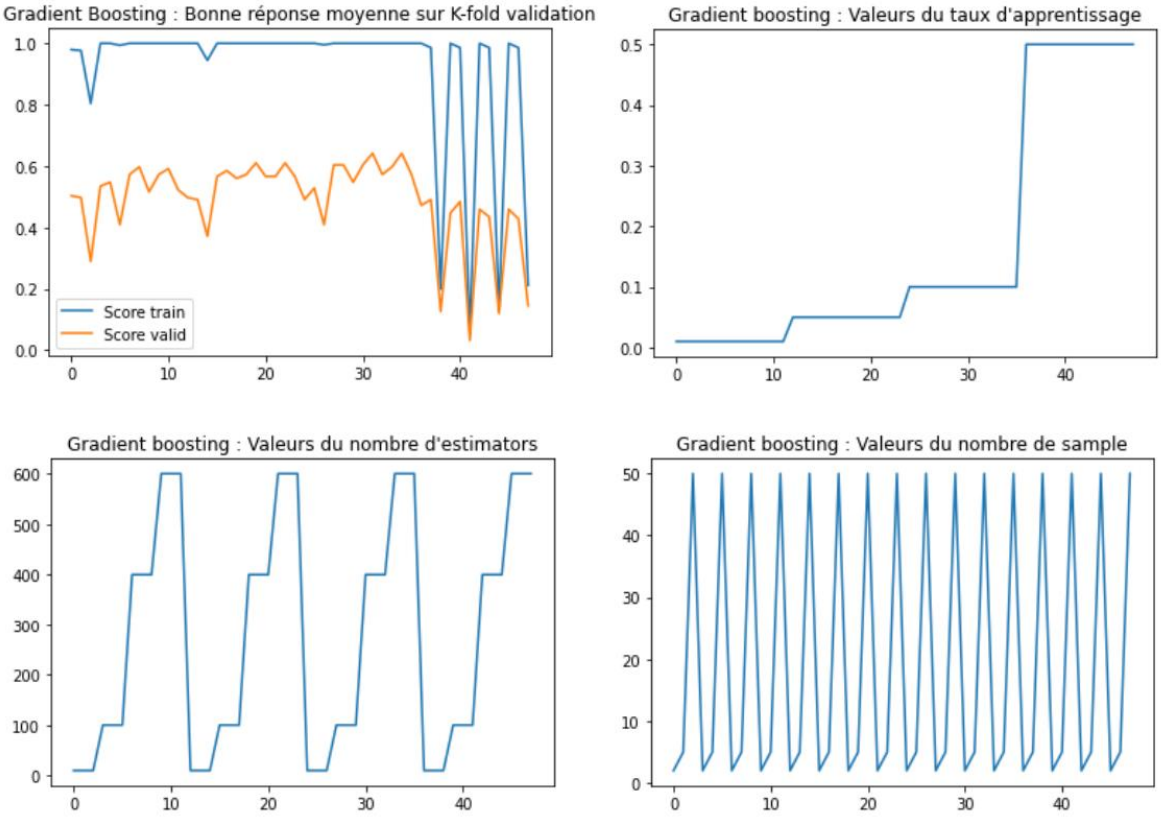
Pour ce modèle, nous avons réalisé des tests sur les différentes valeurs suivantes, pour chaque paramètre :

Learning rate : 0.01, 0.05, 0.1, 0.5

Estimator : 10, 100, 400, 500

Sample : 2, 5, 50

Nous avons ainsi obtenu ces premiers résultats :



Ce programme a alors séléctionné les valeurs suivantes que nous avons implémentées comme valeurs par défaut :

learning_rate = 0.1 *nb_estimator* = 400 *min_sample* = 5

Le paramètre le plus déterminant dans ce modèle semble être le taux d'apprentissage, qui, dès un seuil dépassé peut faire grandement baisser le score si celui-ci n'est pas compensé par les autres paramètres. L'autre paramètre semblant affecté grandement le score est le nombre d'estimators. Celui qui determine combien d'éléments est nécéssaire à la séparation d'un nœud en de nouvelles feuilles.

Nous remarquons cependant un important décallage entre les résultats obtenus avec le dataset de Train et le dataset Test. En effet, un important sur-apprentissage apparait. Nous allons, par conséquent, analyser ce dernier plus en détail dans la partie suivante.

Concernant la quantité d'hyperparamètres choisie, comme le modèle ne possède pas le paramètre *n_jobs* permettant de choisir sur combien de cœurs de l'ordinateur le programme doit tourner, les calculs sont très long. En effet, le fait d'essayer ces hyperparmètres a demandé un temps de calcul de 12 heures. Si nous avions rajouté un paramètre supplémentaire avec deux valeurs, le temps serait multiplié par deux. Nous avons donc été principalement limité dans nos recherches par le coût de calcul.

4. Linear Discriminant Analysis

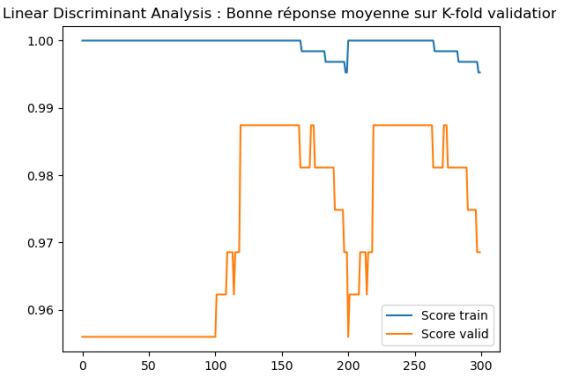
Ce modèle est un classifieur qui adapte une densité gaussienne à chaque classe, en supposant que toutes les classes partagent la même matrice de covariance.

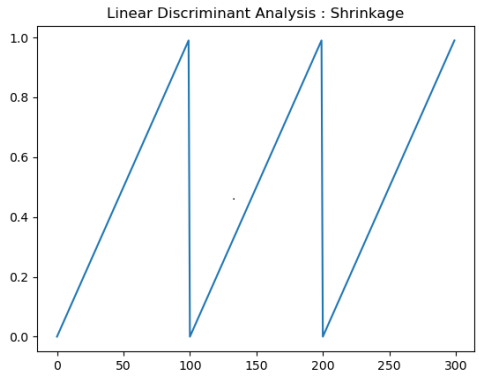
Pour ce modèle, nous avons réalisé des tests sur les différentes valeurs suivantes, pour chaque paramètre :

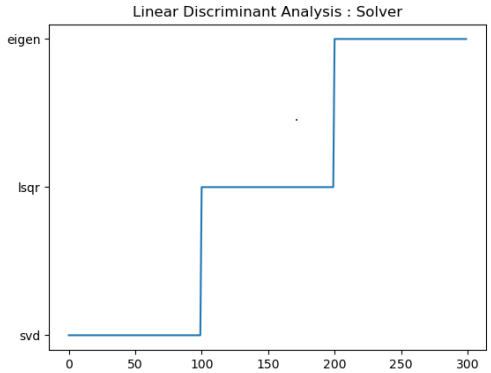
Solver : 'svd', 'lsqr', 'eigen'

Shrinkage : np.arange(0, 1, 0.01)

Nous avons ainsi obtenu les résultats suivants :







Ce programme a alors séléctionné les valeurs suivantes que nous avons implémentées comme valeurs par défaut :

solver = 'lsqr' *shrinkage = 0.05*

Les deux paramètres ont autant d'importance puisque qu'ils se compensent l'un l'autre lors du traçage du score.

Le modèle n'ayant pas de paramètre n_jobs, les calculs peuvent être très longs. Nous avons donc limité notre recherche d'hyperparamètres à ces deux valeurs.

5. SVC

Ce modèle de classification vectorielle est basé sur libsvm. Son temps d'ajustement évolue au moins quadratiquement avec le nombre d'échantillons et peut s'avérer impossible au-delà d'une dizaine de milliers d'échantillons.

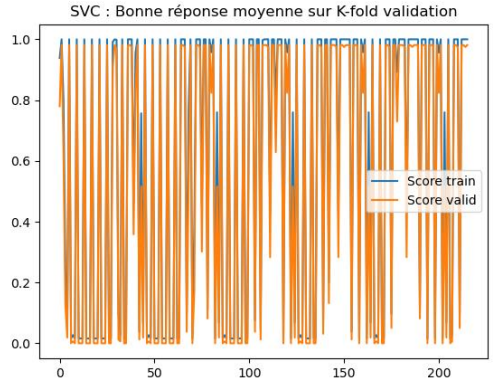
Pour ce modèle, nous avons réalisé des tests sur les différentes valeurs suivantes, pour chaque paramètre :

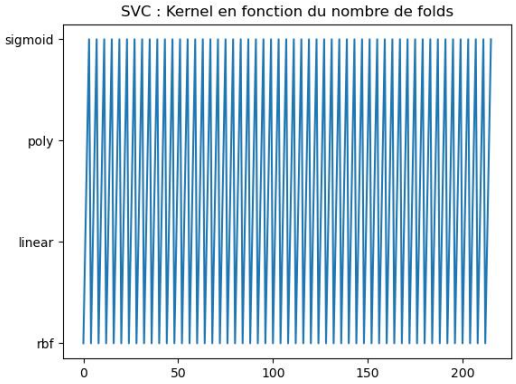
C : 1, 10, 100, 1000, 10 000, 100 000

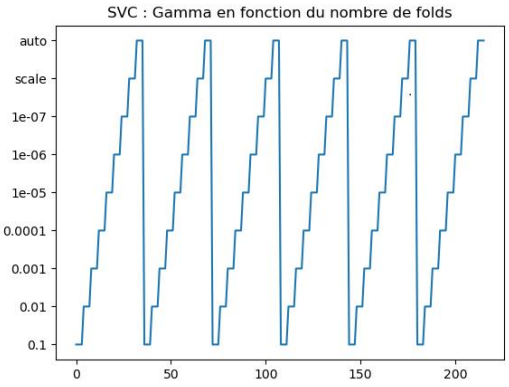
Kernel : 'rbf', 'linear', 'poly', 'sigmoid'

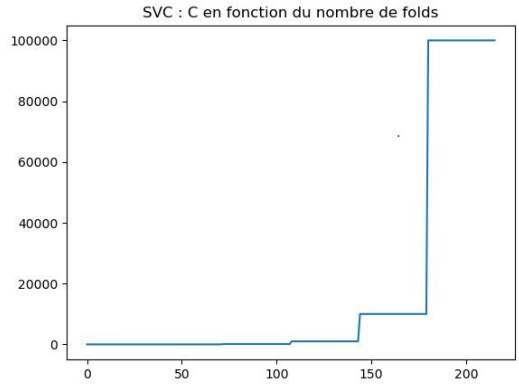
Gamma : 'scale', 'auto', 0.1, 0.01, 1e-03, 1e-04, 1e-05, 1e-06, 1e-07

Nous avons ainsi obtenu les résultats suivants :









Ce programme a alors séléctionné les valeurs suivantes que nous avons implémentées comme valeurs par défaut :

C : 10 *kernel* = 'rbf' *gamma = 'scale'*

Les paramètres les plus déterminants sont gamma et kernel puisque le score suit la forme de ces deux paramètres.

Le modèle n'ayant pas de paramètre n_jobs, nous avons limité notre recherche d'hyperparamètres à ces deux valeurs afin de ne pas avoir de trop longs calculs.

6. NuSVC

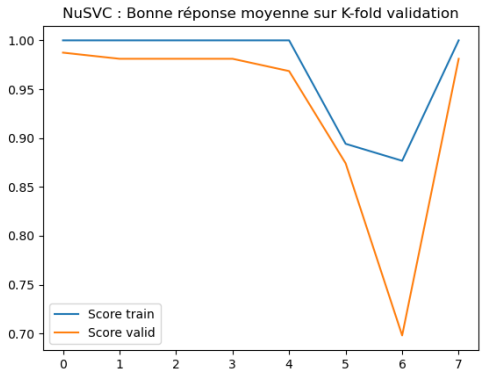
Ce modèle de classification vectorielle est semblable à SVC. Cependant, il utilise un paramètre pour contrôler le nombre de vecteurs de support.

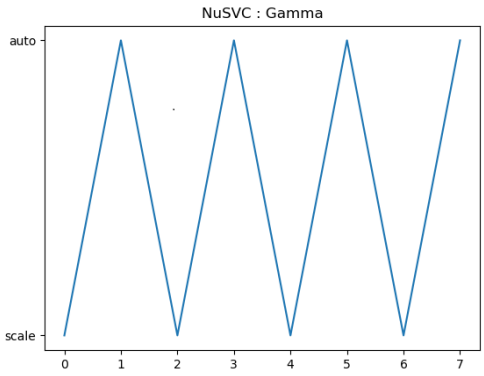
Pour ce modèle, nous avons réalisé des tests sur les différentes valeurs suivantes, pour chaque paramètre :

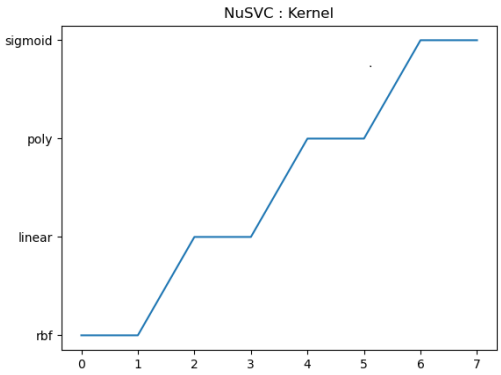
Kernel : 'rbf', 'linear', 'poly', 'sigmoid'

Gamma : 'scale', 'auto'

Et nous avons obtenu les résultats suivants :







Ce programme a alors séléctionné les valeurs suivantes que nous avons implémentées comme valeurs par défaut :

kernel = 'rbf' *gamma = 'scale'*

Le paramètre le plus déterminant dans ce modèle semble être le noyau, qui peut faire grandement baisser le score si celui-ci n'est pas bien choisi.

Le modèle n'ayant pas de paramètre n_jobs, nous avons limité notre recherche d'hyperparamètres à ces deux valeurs afin de ne pas avoir de trop longs calculs.

V / Résultats

Après avoir effectué une recherche des hyperparamètres, nous avons appliqué ces modèles sur le set de donnée Test (pas le fichier !). Cependant, il nous est arrivé d'avoir, sur les données Test, des résultats de score supérieurs aux données Train. Après la réalisation de différents essais avec de la séparation des données aléatoires dans le but de tenter de changer cette tendance, nous sommes arrivés aux mêmes résultats. Ce que nous n'attendions pas à obtenir.

Pour tenter de résoudre ce problème, nous avons également réalisé une boucle similaire à la Kcross-validation mais qui sépare au début du programme le set de donnée Train en 5 sous-set de Train. Puis nous avons tour à tour choisi 4 d'entre eux comme set de données Train et le dernier comme Test. Ceci dans l'optique d'essayer différentes configurations afin de tenter de limiter le risque d'obtenir des résultats plus élevés dans le set Test que dans le Train, qui pourrait être affecté par la génération aléatoire et de la « mauvaise chance ».

Nous avons également testé sur quelques batchs le paramètre N de *feature_selection(N)* avec N=1 et N=0.85. Nous avons alors observé que la différence de résultats entre les deux N est presque constamment de 1% +/- 0.5%.

1. K-Nearest Neighbour

Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
85%	98.61%	97.98%	
100%	98.86%	98.48%	~90%

Séparation test/train numéro	Temps de calcul en secondes	Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
1	0	85%	98.86%	97.98%	
	0	100%	98.36%	96.46%	~90%
2	0	85%	98.99%	97.98%	
	0	100%	99.11%	94.95%	~90%
3	0	85%	99.11%	97.47%	
	0	100%	99.11%	95.45%	~90%
4	0	85%	98.86%	96.97%	
	0	100%	98.48%	98.99%	~90%
5	0	85%	99.24%	97.47%	
	0	100%	98.99%	97.98%	~90%

Avec les meilleurs hyperparamètres choisis, les résultats obtenus sont très bons. Parmi les deux paramètres testés, le nombre de voisins pris en compte est celui qui a le plus gros impact sur le résultat final. Ainsi, plus le nombre de voisins pris en compte est petit, plus le modèle est précis. Cependant, il faut faire attention au sur-apprentissage même si cela ne nous concerne pas selon le IV/1).

2. Random Forest

Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
85%	100%	98.48%	
100%	100%	96.46%	~90%

Séparation test/train numéro	Temps de calcul en secondes	Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
1	2	85%	100%	98.99%	
	2	100%	100%	98.99%	~90%
2	2	85%	100%	94.95%	
	2	100%	100%	95.45%	~90%
3	2	85%	100%	95.45%	
	2	100%	100%	96.46%	~90%
4	2	85%	100%	98.48%	
	2	100%	100%	97.47%	~90%
5	2	85%	100%	95.45%	
	2	100%	100%	97.98%	~90%

Quant au Random Forest, avec les meilleurs hyperparamètres choisis, les résultats obtenus sont également bons. Concernant l'utilisation des paramètres, le nombre d'arbres dans la forêt est important pour la globalité des résultats. La moyenne des résultats avec un nombre d'arbres inférieur à 200 est de 90% alors que si on augmente le nombre, le résultat moyen augmente jusqu'à 95-96%. Cependant, nous pouvons remarquer qu'entre 300 et 800 arbres, le résultat moyen n'augmente que très peu (~0.5%). Le choix de l'hyperparamètre dépend, par conséquent, principalement de la capacité de calcul dont on dispose et du temps durant lequel nous sommes prêts à attendre selon le résultat voulu. Nous aurions donc pu choisir 800 arbres, néanmoins nous avons mis comme valeur par défaut 350.

Nous pouvons également observer, grâce au graphique, que la profondeur des arbres à une importance particulière étant donné que lorsque la profondeur est trop basse, les résultats diminuent grandement comparés à une valeur « convenable ».

Les autres paramètres influent le résultat mais ne semblent pas aussi déterminants que les deux précédents.

Enfin, les données représentées dans la capture d'écran IV/2) ainsi que le tableau plus haut dans les données sont anormalement élevées. En effet, bien que les scores sur le set Test soient à environ 97%, le score constant de 100% sur le set Train nous semble anormal. Même après plusieurs autres essais, nous arrivons à ce résultat.

3. Gradient Boosting

Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
85%	63.66%	69.69%	
100%	62.89%	69.69%	~60%

Les premiers éléments que nous remarquons dans le modèle du Gradient Boosting, sont les résultats inférieurs comparés aux deux précédents modèles.

Le principal hyperparamètre qui caractérise ce modèle est le taux d'apprentissage ou *learning_rate* qui influe beaucoup le résultat. Nous retrouvons par exemple un taux très faible de réussite (20%) lorsque ce taux est trop haut (>0.1).

Autrement, le paramètre *estimator_number* ne semble pas avoir autant d'influence sur le résultat. Néanmoins, un nombre supérieur, aux alentours de 400-500 semble meilleur qu'une valeur de 100.

Nous retrouvons alors sur le set Train des résultats toujours inférieurs à ceux du set Test. Ce qui est dû au hasard de la répartition des données. Pour cela, nous avons également appliqué la recherche de la kcross-validation par différentes configurations de répartitions de subsets Train et Test :

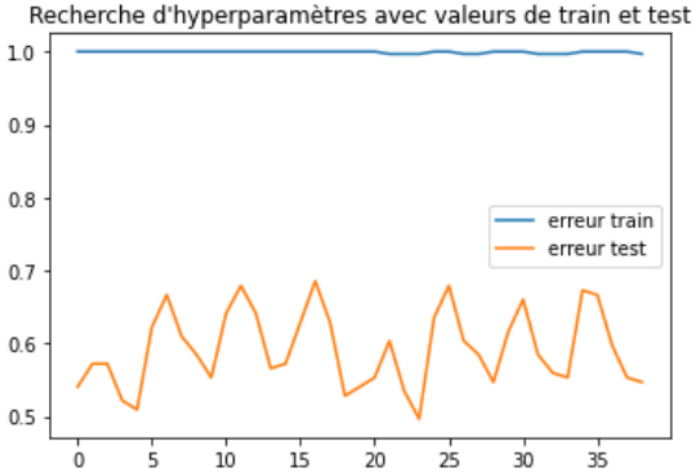
Nous n'avons cependant pas mis tous les résultats de la kcross-validation de la séparation des données d'entrainements pour plusieurs raisons (le seul est ci-dessous). En effet, nous avons essayé de lancer le calcul avec 5 subsets de train/test différents. Dès la première configuration, les résultats obtenus sont, en général, une valeur de bonne réponse avec Train de 100%. Ces résultats sont évidemment du sur-apprentissage au vu des 60% des résultats précédemment observés. Nous avons alors voulu tenter de réaliser une seconde fois la recherche d'hyperparamètre - et ne plus utiliser ceux trouvés meilleurs dans la partie précédente - mais en la combinant cette fois-ci avec la kcross-validation de multiples subsets d'entrainement/test. Nous avons obtenu ces résultats sur la première version des configurations des subsets sur 5 au total.

Pour ce modèle, nous avons réalisé des tests pour chaque paramètre suivant :

Learning rate : 0.05, 0.1

Estimator : 50, 200, 600, 1000

Samples minimum pour la séparation des données : 2, 5, 16, 50, 99



Nous remarquons donc qu'effectivement, le score est presque constant à 100% avec le set Train, mais les résultats sont bien plus faibles sur le set Test. Ce qui montre un important sur-apprentissage. Il est également important à noter que malgré plusieurs tests sur plusieurs ordres de grandeurs comme hyperparamètres, aucun ne nous a permis de résoudre ce problème et d'avoir des résultats plus proches sur les sets de Train et Test.

De plus, nous avons obtenu ce résultat avec 1 configuration de Train/Test sur les 5 possibles avec un peu plus de 12h de calculs. Aussi, nous avons choisi de ne pas réaliser les 4 autres configurations possibles car cela aurait demandé beaucoup trop de temps de calcul.

4. Linear Discriminant Analysis

Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
85%	98.81%	96.62%	
100%	100%	97.47%	~95%

Avec les meilleurs hyperparamètres choisis, les résultats obtenus sont très bons. Parmi les deux paramètres testés, les deux ont un impact équivalent sur le résultat final. Ainsi, plus le shrinkage est élevé, plus le résultat est meilleur.

Ce modèle ne montre également pas de tendance au surapprentissage car bien que les valeurs de score de Train soit élevées, celles de Tests sont également proches.

5. NuSVC

Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
85%	98.85%	97.89%	
100%	100%	98.99%	~90%

Avec les meilleurs hyperparamètres choisis, les résultats obtenus sont très bons. Parmi les deux paramètres testés, le nombre de voisins pris en compte est celui qui a le plus gros impact sur le résultat final. Ainsi, plus le nombre de voisins pris en compte est petit, plus le modèle est précis.

6. SVC

Valeur de l'argument 3 (% max de caractéristiques sur 10%)	Résultats sur le set Train	Résultats sur le set Test	Résultats sur Kaggle
85%	98.61%	97.48%	
100%	98.99%	98.98%	~85%

Avec les meilleurs hyperparamètres choisis, les résultats obtenus sont très bons. Parmi les deux paramètres testés, le nombre de voisins pris en compte est celui qui a le plus gros impact sur le résultat final. Ainsi, plus le nombre de voisins pris en compte est petit, plus le modèle est précis.

Cependant, bien que ce modèle possède des valeurs parmi les plus hautes des six modèles que nous avons, sans pour autant atteindre les 100% se le set de Train, il possède également les valeurs les plus basse si les paramètres sont mal choisis. En effet, on peut dire grâce à la capture d'écran de ce modèle dans la partie précédente, que presque la moitié des valeurs testé ont un score de 0 ou du moins sont très proche. C'est donc un modèle dont on pourrait qualifier les résultats de volatile.

VI / Comparaison des algorithmes

Parmi les six algorithmes étudiés, celui qui a les meilleurs scores est le K-nearest neighbour avec ses 97% de bon résultats sur le set Test. Pour autant, l'algorithme Random Forest possède lui aussi des résultats honorables à 3 ou 4% de moins tout comme le modèle NuSvc et SVC si les paramètres sont bien choisis.

Le seul algorithme ayant des résultats faibles est le Gradient Boosting. Cependant, si nous avions pu effectuer les calculs de cet algorithme sur tous les cœurs de nos ordinateurs nous aurions divisé le temps de calcul d'un facteur 8-10. Nous aurions pu, également, essayer d'autres paramètres avec différentes valeurs. Et nous aurions, par conséquent, pu sûrement augmenter le score de quelques pourcentages.

En ce qui concerne le sur-apprentissage, nos modèles ne semblent pas trop touchés de manière générale étant donné que la différence entre le score du dataset Train et Test n'est pas trop élevée à l'exception du Gradient Boosting, qui lui, observe presque une différence de 50% sur le score moyen entre les deux datasets. Néanmoins, il est possible que le changement d'autres paramètres de ce modèle aurait pu influencer ce facteur. Nous pouvons noter également que beaucoup de score atteignent les 100% de score.

Ainsi, les résultats des algorithmes de Random Forest, LDA et Gradient Boosting ont cependant sur le set de Train, des valeurs très élevées qui s'approchent ou égalisent, une valeur du score de 100%. En plus d'être un élément que nous n'avons pas observé lors des TPs, il nous semble surprenant que de tels résultats soient aussi fréquents (tableaux de la partie V), surtout que les résultats de Kaggle tournent autour des 90% sur ces mêmes modèles.
Aussi, malgré notre tentative de générer différents set Train et Test et de les combiner pour éviter l'effet « chance aléatoire », nous avons tout de même obtenu ces résultats.

VII / Conclusion

Tous les algorithmes que nous avons vus ont des résultats meilleurs que le score moyen sur Kaggle, ce qui montre qu'a priori la plupart des propositions que nous avons mises en place fonctionnent.

Pour ce qui est des résultats excellents sur le dataset Train, il est évident que la qualité des données originales fournies par Kaggle facilite grandement une séparation des données propre. En effet, grâce aux plots réalisé en début de rapport, on peut voir qu'il n'y a pas de pic de données extrêmement loin de la moyenne comme pourraient en laisser des outliner importants.

Enfin, en ce qui concerne les hyperparamètres, nous aurions pu pousser un peu plus loin la recherche et les calculs pour ne pas avoir 100% aussi régulièrement sur le dataset Train mais en contrepartie mieux trier les données sur le dataset Test.

Pour conclure, ce projet n'a pas été vraiment complexe étant donné la qualité des données initiales. Nous aurions en effet dû, si nous avions choisi un autre dataset, passer beaucoup plus de temps sur le traitement des données pour les rendre plus facile à interpréter pour nos modèles.