

Plankton classification - Proposal

Team PLANKTON

Alexandre BESTANDJI

Carlo Elia DONCECCHI

Katia FETTAT

Ugo NZONGANI

Ramdane MOULOUA

Romain MUSSARD

<https://github.com/adblackx/PLANKTON>

<https://codalab.lri.fr/competitions/623>

Semestre 4, 2020

Introduction

Dans le cadre de l'unité d'enseignement *mini-projet*, nous avons choisi de travailler sur le projet **GaiaSavers**. Ce projet a pour objet l'applications de diverses techniques de classification au problème de la reconnaissance d'images, des photos de plusieurs espèces marines. Pour ce faire, nous utilisons la base de données *Bering Sea dataset*. Ces données sont à répartir en 7 classes :

- *chaetognatha*
- *copepoda*
- *euphausiids*
- *fish larvae*
- *limacina*
- *medusae*
- *others*

Nous travaillons en langage *python* à l'aide des modules *pandas*, *seaborn* et *sklearn*. Nous détaillerons le travail effectué en 3 parties :


Preprocessing : Le preprocessing consiste en la préparation des données à l'entraînement.

Cette étape est essentielle au bon déroulement de l'apprentissage, car le format de certaines données pourraient fausser l'apprentissage. Il s'agit aussi de réduire la quantité de données à traiter par les modèles, afin d'économiser du temps de calcul, mais aussi afin d'avoir un ensemble de données le plus représentatif possible afin d'augmenter le score du modèle choisi.

Modèle : L'étape de choix du modèle est cruciale dans l'obtention des meilleurs performances de classification. Elle consiste en une comparaison méthodique de plusieurs modèles existants de classification afin de déterminer lequel est le plus performant à traiter les données.

Visualisation : Il est nécessaire, une fois l'algorithme entraîné, d'avoir un moyen d'évaluer humainement le résultat de la classification, ainsi que d'interpréter ses erreurs afin d'en améliorer l'implémentation.

Metric : Pour tester la performance des différents modèles, nous avons à notre disposition plusieurs outils, dont notamment la metric qui va déterminer notre façon de faire les scores. En effet la metric sert à calculer le score en calculant la "distance" entre le modèle et les données. La metric choisie est la "balanced accuracy metric" qui a l'avantage de ne pas donner trop d'importance aux différentes classes en leur donnant un poids équivalent. En effet selon l'énoncé



nous avons toutes les classes qui sont parfaitement équilibrées, cela équivaut donc à simplement calculer la précision du score, mais si le test est modifié et n'est plus équilibré, la "balanced accuracy score" ne fonctionne toujours correctement tandis que le score de précisions ne le sera pas.

Analyse

Preprocessing

Alexandre BESTANDJI & Romain MUSSARD

Features selection

Dans cette section, nous détaillons les techniques que nous avons employées afin de préparer les données à l'entraînement de l'algorithme de classification.

Le *Bering Sea dataset* nous fournit une banque de 10752 données issues d'images formatées à 100x100 pixels et passées en noir et blanc. Chaque donnée est définie sur 203 features représentant chacune une moyenne de clarté des pixels sur un axe de l'image. Il y a aussi une feature pour la moyenne globale de clarté des pixels, sa variance et la longueur du contours.

Dans un premier temps, et dans l'optique de réduire le temps de calcul des modèles et de maximiser la qualité de nos données, nous procédons à une sélection des features en les évaluant selon le coefficient de corrélation pour ne garder que les features les plus inter-dépendantes, car les features fortement corrélés auront généralement le même effet sur nos données. Il est donc peut intéressant de les conserver.

Afin d'optimiser le nombre de features à utiliser, on trace en suite dans la figure le score en fonction du nombre de features utilisées.

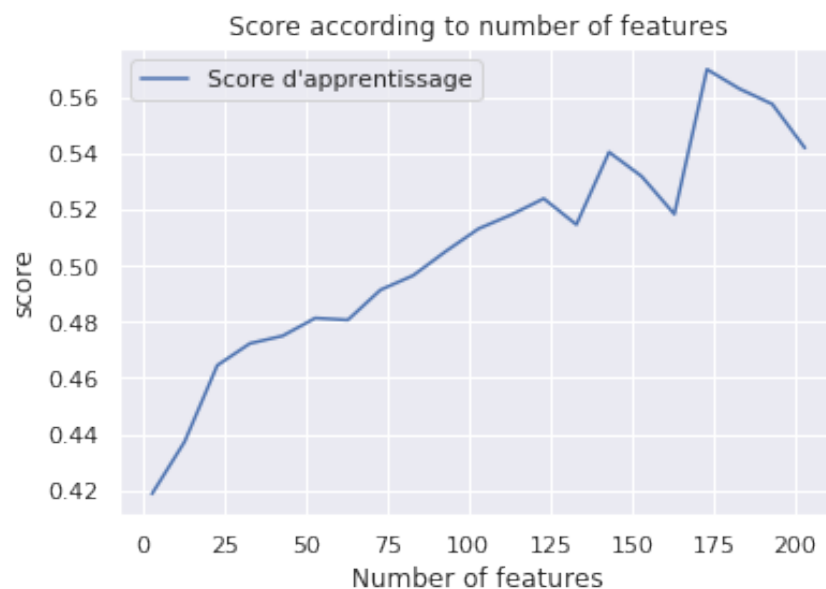


FIGURE 1 – Score en fonction du nombre de features, pour la technique de sélection de features

On remarque un pic d'efficacité autour d'une quantité de 175 features. Cependant, au vu des fluctuations du score pour un nombre de features alentour, il n'est pas impossible que ce pic s'explique par un simple effet de variance.

Nous nous occupons dans un second temps de préparer les données à l'aide d'une technique de préprocessing qu'est la réduction de dimensionnalité par Analyse des Composantes Principales (PCA).

En bref, il s'agit d'extraire les vecteurs propres de la matrice de covariance dans le but d'identifier les features qui influence le moins la variance des données, et ultimement de supprimer ces features en projetant les données sur les features conservées. En d'autres termes la PCA tente de trouver des corrélations entre features et de les exprimer par une formule mathématique afin d'avoir non plus plusieurs centaines de features mais seulement quelques unes.

Après application de la méthode PCA fournie par la bibliothèque *sklearn.decomposition*, on visualise ses effets sur la figure :

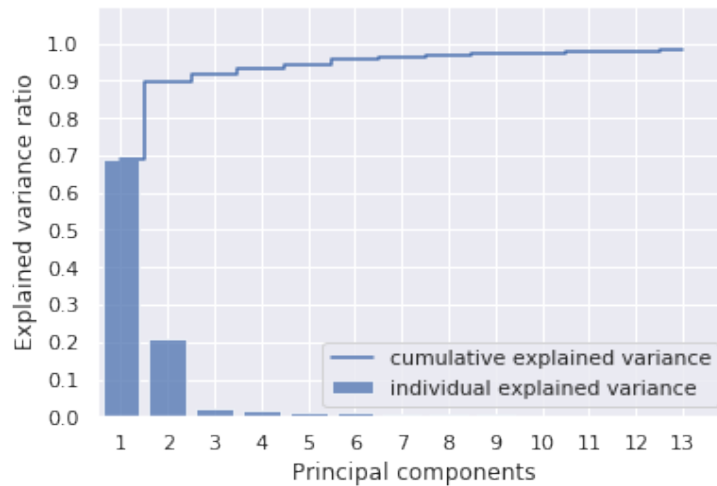


FIGURE 2 – Composantes principales de la variance

On voit ici que 90% de la variance est expliquée par deux composantes principales. Au delà d'une dizaine de composantes principales, on peut déjà expliquer la quasi-totalité de la variance. Voyons donc si le score, calculé de la même façon que précédemment, est augmenté par l'ajout de cette technique, et pour quel nombre de features. On applique ici la méthode PCA et on visualise l'évolution du score sur la figure :

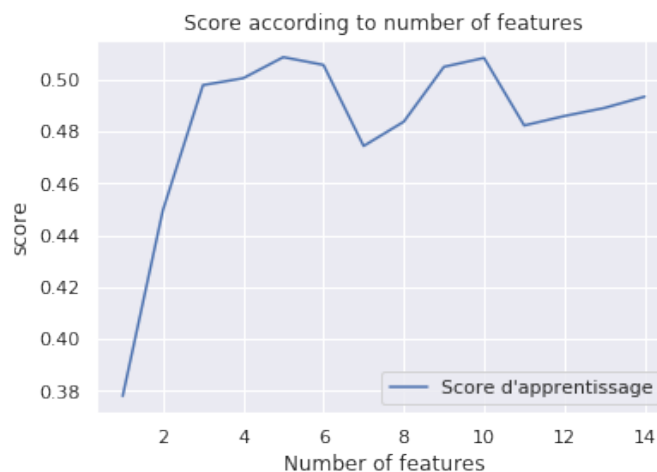


FIGURE 3 – Score en fonction du nombre de composantes principales

On remarque ici que, au delà de 3 composantes principales, le score devient assez difficile à interpréter. On remarque tout de même des pics d'efficacité pour 5 et 10 features, mais les différences de scores avec d'autres nombre de features sont assez peu significatives. Une fois de plus, il pourrait être utile d'étudier la variance de tels résultats afin d'y voir plus clair.

Dans la continuité de ce projet, nous nous proposons de tester d'autres techniques de sélections de features, comme la méthode selectKBest de scikit learn en utilisant le test statistique du χ^2 .

Finalement nous avons choisis de créer une classe Preprocessor qui utilisera une combinaison de technique de preprocessing. Plus particulièrement nous avons choisis de travailler avec selectKbest et PCA en ce qui concerne le traitement des features.

Nous avons cependant essayé d'autre méthode comme la selection de features en utilisant les coefficients de corrélation mais cette technique était moins efficace que selectKbest. Nous avons fait deux fonctions afin de sélectionner les meilleurs méta-paramètre pour la pca et pour le selectKBest.

Pour ce faire on fait varier les méta-paramètres tout en faisant des cross-validations successive. on gardera les méta-paramètres qui renvoient les meilleurs scores. Pour classer les données et faire la cross validation nous n'utilisons pas notre modèle mais une Forêt d'arbres décisionnels pour ne pas avoir un temps de calcul trop long.

Outliers detection

Nous nous occupons en suite de trier les valeurs aberrantes (Outliers) des données. Cette opération est importante car certaines données mal labélisées ou ambiguës pourraient fausser l'apprentissage. On utilise la méthode d'écart interquartile (IQR) qui mesure la dispersion des données. Pour une visualisation correcte, on représente les données et leurs valeurs aberrantes selon 2 composantes principales, donc après PCA, comme visible dans la figure suivante .

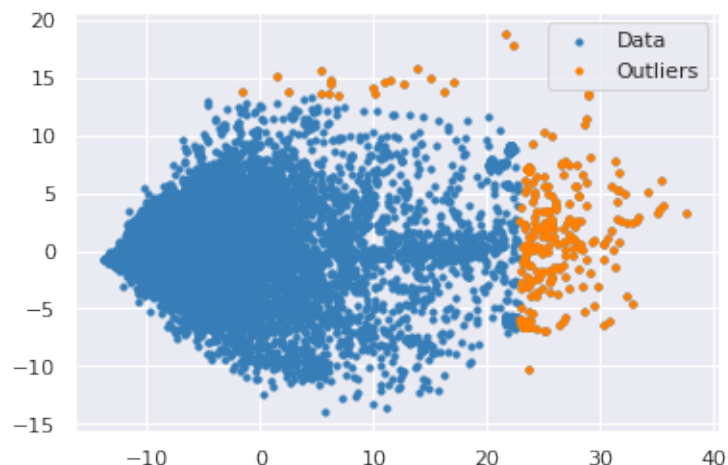


FIGURE 4 – Mise en évidence des outliers grâce à l'IQR

Cette méthode n'arrive cependant pas à améliorer le score. Celui-ci passe de 0.436 avant à 0.388 après le traitement. On essaie donc une autre technique de suppression des valeurs aberrantes, appelée LocalOutlierFactor de la bibliothèque *sklearn.neighbors*. On peut visualiser le type de sélection appliquée sur la figure .

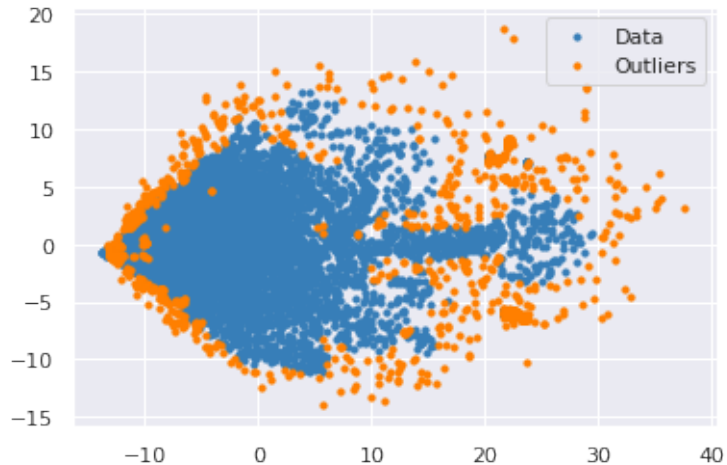


FIGURE 5 – Mise en évidence des outliers par LocalOutlierFactor

On remarque que les valeurs sont triées de façon à ce que les outliers soient les données "limites" de l'espace des features. On peut alors tracer un graphe du score en fonction du nombre de voisins des données.

On remarque que la technique fait augmenter sensiblement le score, et on en conclut qu'elle est préférable à la précédente pour notre jeu de données.

Nous avons choisi dans notre projet de détecter les données aberrantes avec la fonction Local Outliers Factor de scikit learn. Nous avons testé d'autres techniques de détection en utilisant le z-score ou encore l'IQR mais nous avons choisi de garder Local Outliers Factor pour maximiser le score. Comme pour le traitement des features nous avons fait une fonction qui cherche le nombre de voisins optimal à utiliser en nous aidant de la cross validation sur l'ensemble d'entraînement. La technique de suppression des outliers semble être celle qui donne de meilleurs résultats sur le score.

Bonus :

Le tableau ci-dessous est obtenu après suppression des outliers, suivi d'une sélection des 197 meilleurs features et d'une PCA avec 8 composantes principales.

Dataset	Num. Examples	Num. Variables/features	Sparsity	Has categorical variables ?	Has missing data ?	Num. examples in each class
Training	9676	8	0	0	0	1248 to 1474
Valid(action)	3584	8	0	0	0	512
Test	3584	8	0	0	0	512

FIGURE 6 – Statistiques sur les données

Conclusion du Preprocessing :

Dans notre preprocessing nous avons finalement choisit de faire de la sélection de features avec selectKbest et le test du χ^2 puis de l'associer à de la réduction de dimension avec de la pca et de supprimer les outliers ou données abérantes avec la technique de Local Outliers Factor.

Afin de trouver les meilleurs méta-paramètres pour chaque technique de preprocessing utilisé nous avons fait des fonction de recherche des meilleurs méta-paramètres par dichotomie.

C'est ainsi que nous avons trouvé que le score sera maximal en gardant les 197 meilleurs features avec selectKbest, en faisant une PCA avec 19 composantes et en tenant compte des 3 voisins les plus proches dans la PCA.

Cependant le score obtenu sera plus bas avec preprocessing que sans. Pour augmenter le score avec le preprocessing il devient nécessaire de travailler sur les données brut.

IMPORTANT : Pour avoir une idée plus compète de l'étendu du travail réaliser pour le pré-processing se référer au jupyter notebook https://github.com/adblackx/PLANKTON/blob/master/starting_kit/README_Preprocessing.ipynb

Model

Katia FETTAT & Ramdane MOULOUA

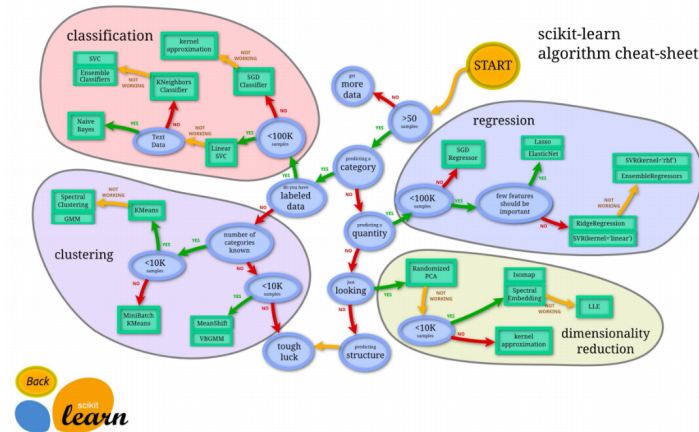


FIGURE 7 – Les modèles [14]

Dans cette section nous nous intéressons au choix du meilleur modèle. Nous nous sommes intéressés aux modèles de classification donc les "labeled data", nous avons testé les modèles en suivant le schéma mais aussi ceux de la documentation. Nous avons travaillé sur les données qui ont déjà été pré-traitées (mais seulement sur celles-ci) pour déterminer le meilleur modèle. En local nous avons les ensembles de train, et en ligne nous avons l'ensemble de Valid.

		Classe réelle	
		-	+
Classe prédite	-	True Negatives (vrais négatifs)	False Negatives (faux négatifs)
	+	False Positives (faux positifs)	True Positives (vrais positifs)

FIGURE 8 – Les matrices de confusion

Remarquons que nous pouvons aussi utiliser une matrice de confusion dans la page jupyter pour regarder la précision du modèle, le but étant que la matrice ait une diagonale qui devrait le plus possible se rapprocher de 1 (si on a une matrice d'identité on over-fit, si les valeurs des diagonales sont petites, alors on a un under-fitting). La matrice de confusion aide beaucoup à voir la précision d'un modèle, notamment à voir si une classe pose problème par exemple.

Par la suite, nous avons fait le score avec la "balanced metric" sur le train et le valid data. Nous nous sommes rendu compte que le score était correcte se rapprochant de 1, pouvant donner un score (pour le meilleur modèle qui est RandomForestClassifier avec les meilleurs paramètres selon la fonction RandomizedSearchCV) de 0.95 de metric avec la balanced accuracy score pour le train (ce qui très performant). La cross-validation est aussi bonne avec 0.78, et la validation et 0.73 en ligne ce qui est correcte.

La cross-validation est le fait que l'on divise nos données de train en plusieurs parties d'entraînement et test, ce qui permet d'évaluer le modèle et voir s'il est bon, s'il over-fit ou s'il

	Model	Cross-Validation	train
0	Nearest Neighbors	0.708810	0.999814
1	Decision Tree	0.614288	0.726656
2	Random Forest	0.779545	0.999814
3	Neural Net	0.510780	0.611979
4	AdaBoost	0.404023	0.426990
5	Naive Bayes	0.431674	0.437500
6	QDA	0.460282	0.606585
7	ExtraTreesClassifier	0.779715	0.999814

FIGURE 9 – Résultats pour les Différents modèles testés avec la *Balanced_accuracy_metric*

under-fit. Il y a plusieurs typer de cross-validation. Nous avons utilisé le K-fold dans le projet, mais un autre type existe qui est le hold out. Le hold-out divise l'ensemble en deux, le premier sert à entrainer le modèle et le second sert à le valider à l'aide d'une metric. Pour ma k-folder strategy on commence par prendre un élément de l'ensemble, on entraine le modèle sur tout l'ensemble restant et on regarde la performance du modèle sur le premier élément, et on le fait pour chaque donnée du modèle et il nous reste plus qu'à faire la moyenne.

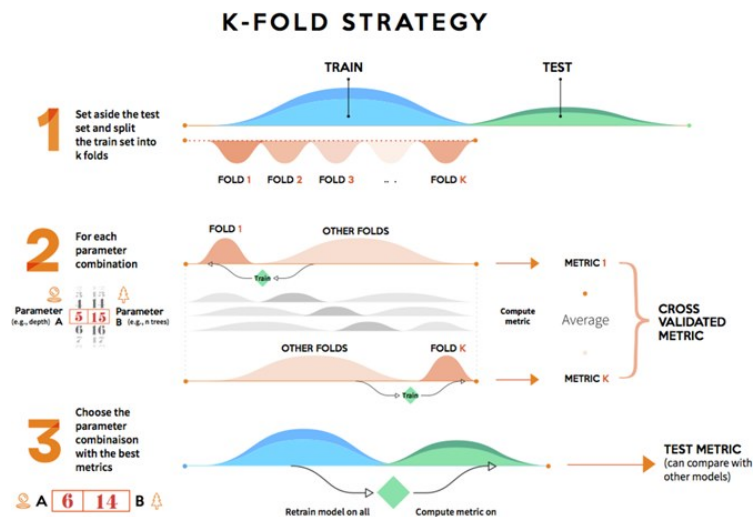


FIGURE 10 – K-Fold Strategy [15]

D'après les résultats, nous avons deux bons candidats pour le modèle, dont ExtratreesClassifier et Random Forest. De plus nous sommes sans pre-processing, il semble pas y avoir d'overfitting pour le train on a 0.77 et d'après les résultats en ligne on a 0.73 pour RandomForest, il n'y a donc pas d'overfitting. Il reste à améliorer le tout en testant le stacking par exemple et en intégrant le pre-processing(ou en testant les meilleurs hyper-paramètres en fonction du pre-processing).

RandomForest a des points forts qui sont détaillés dans notre notebook comme par exemple

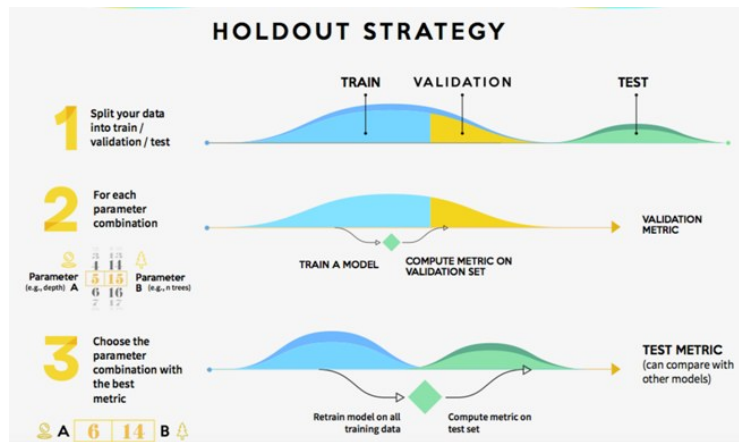


FIGURE 11 – Hold-out Strategy
[15]

que l'algorithme n'est pas biaisé parce qu'il examine l'intégralité des données et il est stable car il n'est pas affecté lorsque de nouvelles données sont introduites, en particulier lorsque les nouvelles données affectent un arbre, l'algorithme fonctionne bien avec des données qui ne sont pas mises normalisées. L'algorithme a une complexité qui peut le rendre lent pour de très grandes données (supérieurs à 100k)

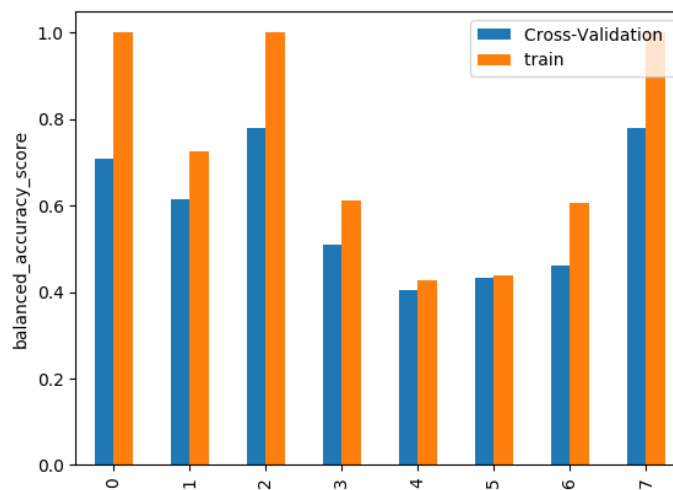


FIGURE 12 – Comparaison par Diagramme en Barre des Différents modèles testés avec la *Balanced_accuracy_metric*

La principale différence entre le stacking et le voting c'est que le stacking examine la probabilité la plus élevée pour une donnée calculée par tous les modèles. Alors que le voting examine le plus grand nombre d'allocations de données de tous les modèles, par exemple, si trois modèles sur 4 choisissent la fonctionnalité 1, ce sera la fonctionnalité 1.

Nous avons travaillé sur le voting. En effet, nous avons choisi les trois meilleurs modèles ainsi que les meilleurs paramètres pour chacun de ces modèles là. Nous avons également essayé de corriger le sur-ajustement des données mais rien n'y fait, nous avons toujours de l'over-fitting, cependant, nous avons de meilleurs score par rapport au RandomForestClassifier qui est tout seul. Cependant il semble être déconseillé de l'utiliser pour la suite du projet, ou bien parce qu'on attends de nous que l'on crée ce modèle.

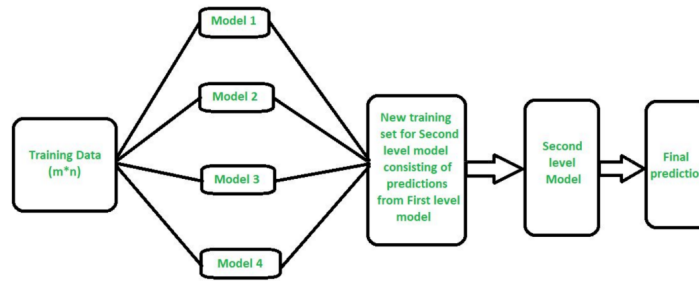


FIGURE 13 – Schéma du Stacking *Balanced_accuracy_metric* [13]

Nous avons associé notre modèle avec le pre-processing à l'aide d'un pipeline.

Liste des modèles testés :

- *Nearest Neighbors*
- *Decision Tree*
- *Random Forest*
- *Neural Net*
- *AdaBoost*
- *Native Bayes*
- *QDA*
- *Extra Trees Classifier*

Conclusion Modèle :

Nous sommes passé aux RAW DATA. Dans le fichier `plkClassifier.py`, la classe `plkClassifier` est la classe est utilisée pour trouver le meilleur modèle (sous forme d'un voting par exemple) à l'aide de `assistModel` et `classCVM`. Où `assistModel` est la classe qui recherche les meilleurs modèles. Pour chaque modele envoyé en paramètre, on trouve les meilleurs parametres à l'aide de `GridSearchCV` ou `RandomSearchCV` (qu'on peut choisir), et on réunit les meilleurs modeles avec leurs meilleurs paramètres en revoyant un voting. Enfin, `Classifier` sert simplement à fit les modèles, faire la metric et la cross validation. Dans `assistModel`, les meilleurs modèles sont choisi à l'aide d'un seuil dans `compareModel` au niveau de la metric et du CV-score. Cependant sur Codalab nous l'avons pas encore soumis tel quel et n'utilions pas `plkAssitClassifier` pour le moment. Les tests sont dans `mainPLK.py` On a atteint notre objectif, c'est à dire d'avoir un modèle qui en fonction des données, va sélectionner les meilleures modèles ainsi que leurs meilleurs hyper-parametres, pour ensuite faire un modèle de voting. Le travail fourni dans le code montre tous les modeles et hyper-parametres testés, ils sont en général testé en envoyant des parametres différents dans modèle à l'aide des dictionnaires.

Visualisation

Carlo Elia DONCECCHI & Ugo NZONGANI

Dans cette section nous détaillons les différentes méthodes que nous avons utilisées pour effectuer la visualisation des données.

Le travail de visualisation s'est fait selon les 4 étapes suivantes :

- Effectuer une première visualisation des données pour avoir une première approche du problème
- Visualiser les clusters dans les données
- Visualiser les erreurs de prédiction
- Visualiser les performances d'un modèle sur nos données

Pour la 1ere étape nous avons trouvé un moyen simple de visualiser certaines caractéristiques des données pouvant nous apporter des informations sur certaines classes de plancton. Nous avons choisi d'effectuer cette visualisation sur les critères suivants : moyenne de pixel, variance et longueur du contour.

Voici ce que nous avons obtenu :

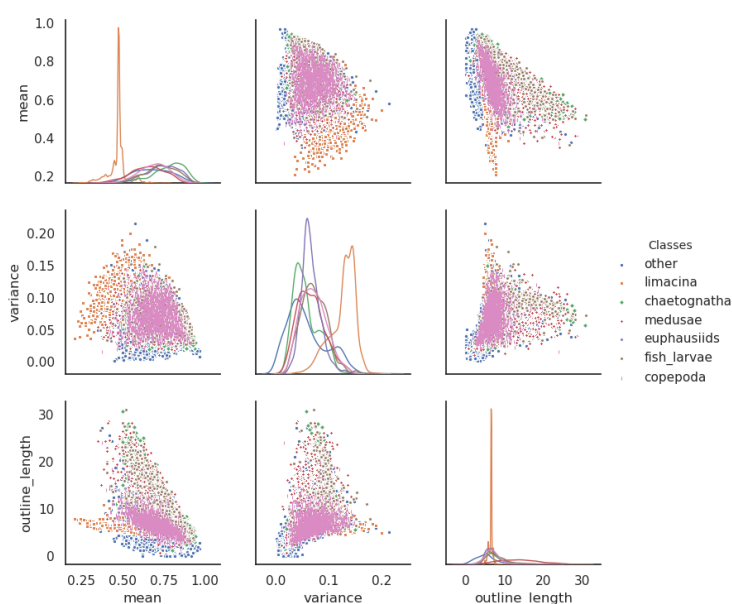


FIGURE 14 – Comparaisons des 7 classes de plancton en fonction des critères : mean, variance et outline length

On observe que la classe *limacina* se distingue fortement des autres par la longueur de ses contours et la moyenne de ses pixels. Ces deux critères seront donc sûrement suffisants pour déterminer le label des images de cette classe. Les autres classes sont quant à elles très similaires au niveau de leur variance, moyenne et longueur des contours.

Grâce à cette figure nous avons vu que les trois critères cités précédemment ne semblent pas être très départageant pour la plupart des classes de notre jeu de données. La moyenne de pixel, variance et longueur de contours des images des planctons des classes différentes de *limacina* semblent être plus ou moins les mêmes.

Nous avons ensuite cherché à visualiser les clusters dans les données, c'est-à-dire voir de quelle façon les différentes classes du jeu de données sont réparties les unes par rapport aux autres. Pour cela nous avons choisis d'utiliser l'algorithme des *k-moyennes* vu en cours. Nos

données sont regroupées en 7 différentes classes correspondants aux différents types de plancton comme précisé précédemment. Nous présentons ci-dessous la figure obtenue.

K-means clustering on the digits dataset (PCA-reduced data)
Centroid are marked with white circles

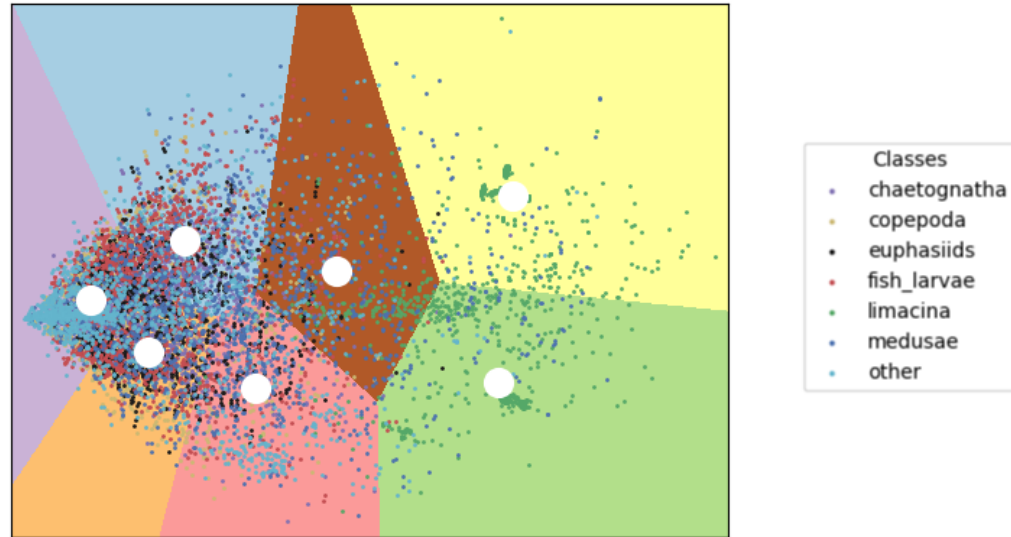


FIGURE 15 – Répartition des 7 classes de plancton

Chaque couleur correspond à une des 7 classes. On remarque tout d'abord que la répartition des classes les unes par rapport aux autres est inégale, effectivement certains centres sont très proches les un des autres alors que d'autres sont bien plus isolés. Concrètement, cela signifie que certaines classes sont plus difficiles à discerner que d'autres. On observe que les 3 classes les plus à gauche sur la figure sont très proches, elles se "ressemblent" donc plus. On remarque également que la répartition des points qui représentent les données est complètement inégale. Certaines zones contiennent d'importants amas de points tandis que d'autres n'en contiennent que très peu. L'information à extraire de cette figure est qu'il ne semble pas vraiment y avoir de correspondance entre les clusters détectés par l'algorithme des *k-moyennes* et les classes naturelles. La représentation des données n'est donc pas bonne. Cependant on voit que les points de la classe *limacina* représentés en vert sont plutôt bien répartis et assez éloignés de tous les autres. Cette observation va dans le sens de ce que nous avons observé avec la figure précédente, cette classe semble être la plus simple des 7 à discerner.

La prochaine étape de notre travail était de trouver un moyen de visualiser les erreurs de prédiction. Pour cela nous avons utilisé la surface de décision. Pour obtenir des résultats variés nous avons utilisé les classifieurs suivants : *DecisionTree*, *RandomForest*, *MLP*, *KNeighbors*, *AdaBoost*, *GaussianNB*, *QuadraticDiscriminantAnalysis* et *ExtraTree* sur nos données. Voici les différents graphiques que nous avons obtenus.

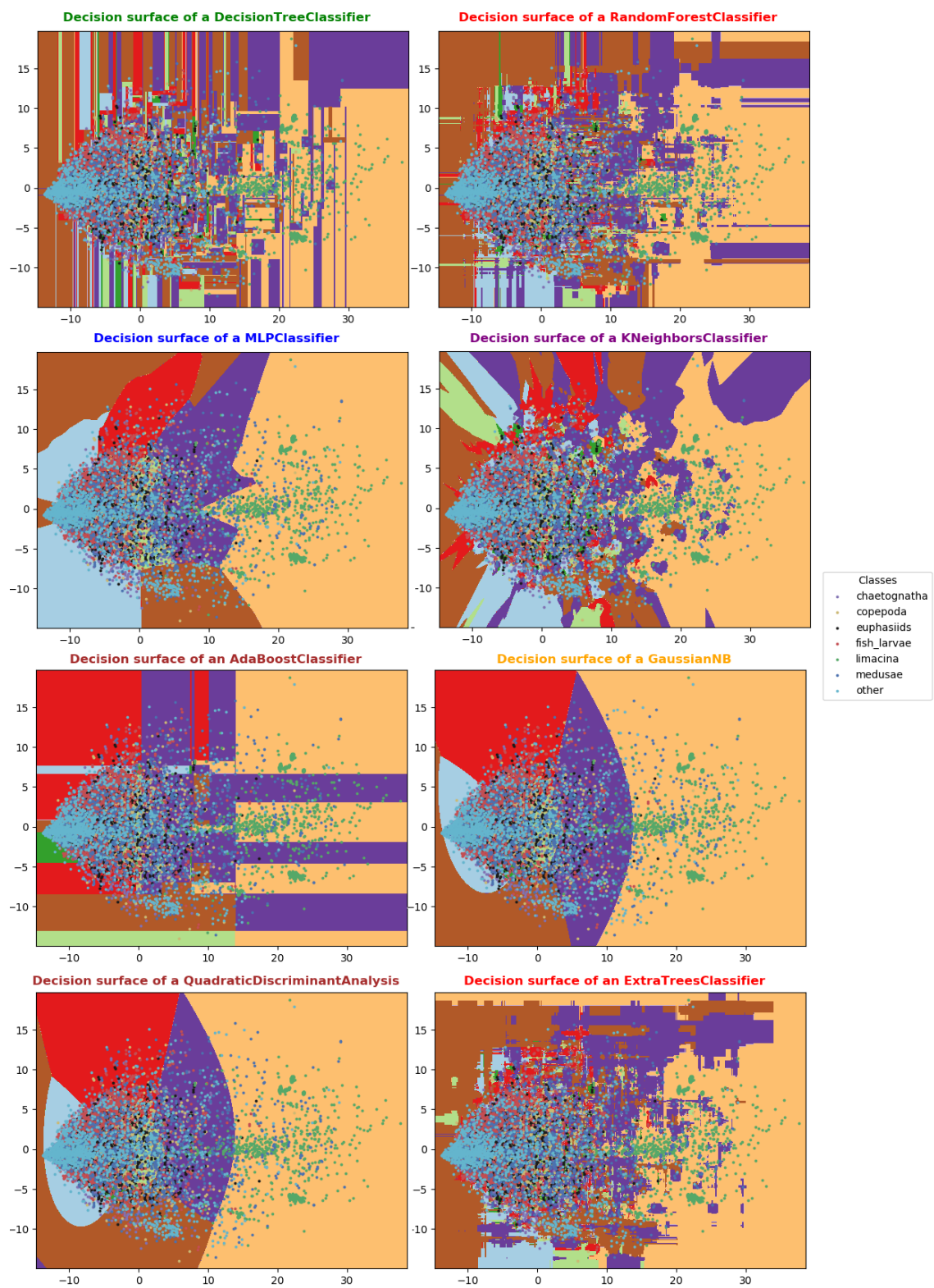


FIGURE 16 – Surface de décision des différents classifieurs

Chaque point sur les figures représente une donnée, c'est-à-dire un plancton appartenant à une des 7 classes indiquée dans la légende et les différentes couleurs de fond indiquent pour cette région la décision prise par le classifieur. On remarque tout d'abord qu'on obtient des résultats plus ou moins différents selon les classifieurs utilisés.

On remarque que les résultats obtenus par le *DecisionTree*, le *RandomForest* et l'*ExtraTrees* se ressemblent beaucoup. Cette similitude s'explique par le fait que le *RandomForest* et l'*ExtraTrees* utilisent tous les deux le *DecisionTree*. Un classifieur *RandomForest* est une 'forêt' d'arbre de décision (decision tree) et un *ExtraTrees* diffère d'un *RandomForest* dans la façon dont il construit ses arbres de décision, il est plus aléatoire. On remarque sur les figures de ces 3 classifieurs la présence de barres verticales et horizontales un peu partout, cela traduit de l'overfitting. Les différents arbres de ces classifieurs se sont trop adaptés aux échantillons de données avec lesquels ils se sont entraînés. Cependant on note qu'il y a moins d'overfitting avec le *RandomForest* et l'*ExtraTrees*, leur performance est également meilleure que celle du *DecisionTree*. On peut expliquer cette différence d'efficacité par le fait que le *DecisionTree* ne contient qu'un arbre de décision qui va effectuer la décision finale, alors que les deux autres classifieurs en contiennent un nombre bien plus important et chacun d'entre eux est différent, il y a donc bien plus de diversité. Les formes que l'on voit sur leur figures sont des barres verticales car à un arbre fait une suite de comparaison *if-then-else* et une seule par feuille puis en fonction du résultat découpe petit à petit le plan en cases avec ces barres. La décision finale est faite selon la majorité, c'est-à-dire en prenant la décision la plus prise par l'ensemble des arbres.

La surface de décision du *KNeighbors* ressemble à celle des 3 classifieurs précédents à la différence que les formes sont courbées, il y a très peu de ligne droite. La forme finale de cette figure est une tessellation. La figure est composée de cellules de Voronoï, "Voronoï cells", chaque point est entouré d'une de ces cellule. Tous les points se trouvant ensuite dans une cellule sont attribués à la classe du point qui se trouvait dans la cellule initialement. Les formes sont plutôt des lignes assez droites car notre *KNeighbors* utilise ici la distance euclidienne, les formes des cellules sont alors des polygones. Les frontières de chaque cellule sont positionnées telles qu'elles sont à distance égale entre les deux points les plus proches. Une couleur est attribuée à chaque cellule en fonction de la classe choisie par le classifieur c'est pour ceci qu'on ne voit pas toutes les cellules car plusieurs de même couleur sont dans la même région. Les résultats semblent plutôt bons pour ce classifieur.

Les résultats de l'*AdaBoost* ressemblent à certains endroits à ceux du *DecisionTree*, *RandomForest* et *ExtraTrees* car on note également la présence de barres verticales et horizontales mais beaucoup plus larges cette fois. Cette ressemblance est liée au fait que l'*AdaBoost* est également une 'forêt', pas d'arbre de décision mais de 'souches' (stumps), ce sont des weak learners car se sont de petits arbres composés seulement d'une racine et de deux feuilles. Chaque stump est réalisé en prenant en compte l'erreur de son prédécesseur. Cependant on voit que les résultats ne sont pas très bons, les décisions prises par *AdaBoost* ne donnent que des régions rectangulaires ne prenant pas en compte les différentes classes de plancton. Les résultats ressemblent donc légèrement à ceux des 3 premiers classifieurs mais sont moins bons car *AdaBoost* est une combinaison de weak learners, des petits arbres moins efficaces pour répondre à notre problème que les arbres plus complets de ces classifieurs. On obtient comme pour nos trois premiers classifieurs des formes horizontales et verticales pour la même raison.

La surface de décision du *MLPClassifier* (Multi-Layer Perceptron) ne donne pas de bons résultats, on peut le voir assez rapidement car les couleurs de fond semblent ne pas tenir compte des différentes classes, on voit par exemple que dans la partie droite de la figure le classifieur ne semble pas faire la différence entre les *limacina* en vert et les *medusae* en bleu, pour le reste

de la figure on observe sensiblement la même chose pour les autres classes.

Les résultats des *GaussianNB* et *QuadraticDiscriminantAnalysis* sont très similaires. Mais seules les décisions pour la classe *limacina* en vert semblent être justes. Parmi la liste des classifieurs que nous avons utilisés, 3 d'entre eux semblent sortir du lot, il s'agit du *RandomForest*, de l'*ExtraTrees* et du *KNeighbors*.

La surface de décision nous permet donc de visualiser rapidement les erreurs de prédictions du classifieur qu'on utilise.

La dernière étape était comme nous l'avons annoncé précédemment de visualiser les performances d'un modèle sur nos données pour pouvoir évaluer visuellement son efficacité. Pour cela nous avons utilisé la fonction *learning_curve* de *sklearn* pour obtenir le score du modèle puis nous avons fait une fonction pour afficher le score d'un modèle avec les barres d'erreurs en fonction de la taille de l'ensemble d'apprentissage.

Voici les résultats obtenus avec le modèle initial proposé au début du projet.

Machine learning model's performance according to the size of the training set

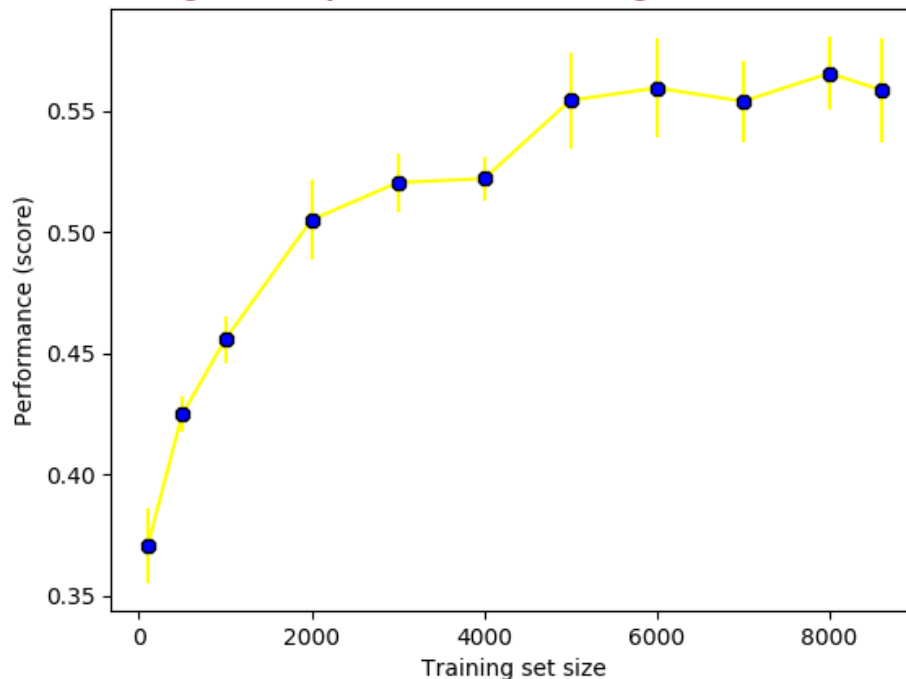


FIGURE 17 – Performance du modèle initial avec les barres d'erreurs

Voici maintenant les résultats obtenus avec le modèle final sélectionné par le groupe *Model*.

Machine learning model's performance according to the size of the training set

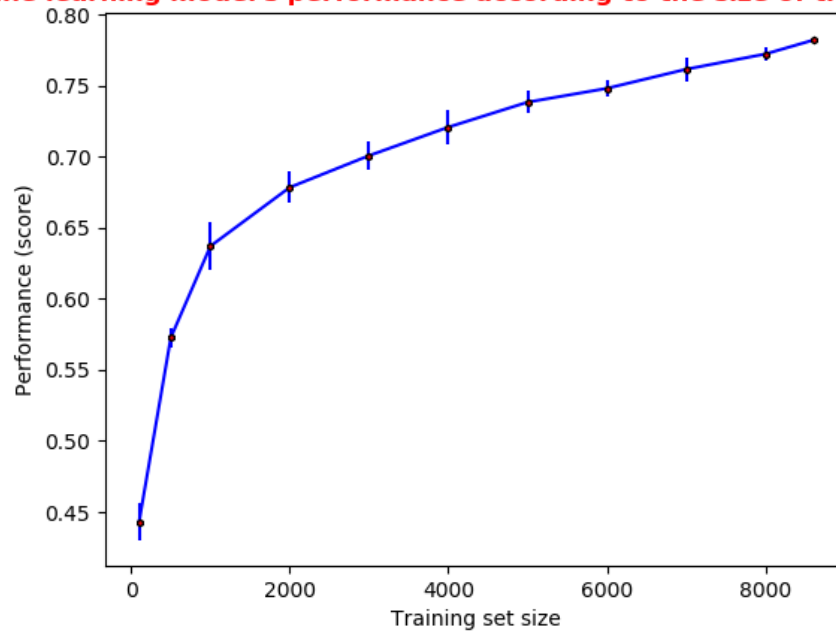


FIGURE 18 – Performance du modèle final avec les barres d'erreurs

On observe tout d'abord que pour le modèle initial le score obtenu était de 0.55 alors que pour le modèle final nous avons un score d'environ 0.78. On remarque que sur les deux figures que plus la taille de l'ensemble d'apprentissage augmente, plus le score obtenu par le modèle est élevé. Lorsqu'on atteint la taille maximale de 8601, on obtient un score d'environ 0,78 pour le modèle final. On voit également que les barres d'erreurs sont relativement petites, l'incertitude sur le score est donc assez faible. De plus, leur taille est décroissante pour le modèle final, ce qui indique que plus le modèle est entraîné par un ensemble d'apprentissage important, moins il fait d'erreur.

Les barres d'erreur sont obtenues par le programme en calculant l'écart-type des résultats de folds de cross-validation. Ces barres mesurent la variabilité de notre ensemble d'apprentissage et de test. Elle ne mesure pas la variabilité de l'initialisation de l'algorithme car le seed de notre *RandomForestClassifier* est fixé. La façon dont l'ensemble d'apprentissage est divisé en folds lors de la cross-validation intervient également dans la barre d'erreur car le mélange du jeu de donnée varie en fonction de la cross-validation, il y a donc une composante aléatoire qui intervient indirectement dans la barre d'erreur.

Pseudo code pour le preprocessing (1):

```
01 |     def __init__(self, n_components = 8, nb_feat = 193, nbNeighbors = 3):
02 |         self.skb = SelectKBest(chi2, k= nb_feat)
03 |         self.pca = PCA(n_components)
04 |         self.lof = LocalOutlierFactor(n_neighbors=nbNeighbors)
05 |
06 |     def fit(self, X, Y):
07 |         '''
08 |             Run score function on (X, Y) and get the appropriate features.
09 |
10 |             Paramaters
11 |             -----
12 |             X : array-like of shape (n_samples, n_features) representing The
training input samples.
13 |             Y : array-like of shape (n_samples,) representing The target values
14 |
15 |             Returns
16 |             -----
17 |             self : object
18 |
19 |         '''
20 |         self.skb = self.skb.fit(X,Y)
21 |         X_temp = self.skb.transform(X)
22 |         self.pca = self.pca.fit(X_temp)
23 |         return self
24 |
25 |     def fit_transform(self, X, Y):
26 |         '''
27 |             Fit to data, then transform it.
28 |             Fits transformer to X and Y with optional parameters fit_params
29 |             and returns a transformed version of X.
30 |
31 |             Paramaters
32 |             -----
33 |             X : array-like of shape (n_samples, n_features) representing the
training input samples.
34 |             Y : array-like of shape (n_samples,) representing The target values
35 |
36 |             Returns
37 |             -----
38 |
39 |             X_new : numpy array of shape [n_samples, n_features_new] representing
the transformed array.
40 |         '''
41 |         return self.fit(X, Y).transform(X)
42 |
43 |     def transform(self, X):
44 |         '''
45 |             Transform the data and returns a transformed version of X.
46 |
47 |             Paramaters
48 |             -----
49 |             X : array-like of shape (n_samples, n_features) representing the
training input samples.
50 |
51 |             Returns
52 |             -----
53 |
54 |             X_new : numpy array of shape [n_samples, n_features_new] representing
the transformed array.
55 |         '''
56 |         X_res = self.skb.transform(X)
57 |         X_res = self.pca.transform(X_res)
58 |         return X_res
59 |
60 |     def outliersDeletion(self, X, Y):
61 |         '''
```

```

62 |         Detect the outliers and return the training input samples (X)
63 |         and the target values (Y) without the outliers
64 |
65 |         Paramaters
66 |         -----
67 |         X : array-like of shape (n_samples, n_features) representing the
training input samples.
68 |         Y : array-like of shape (n_samples,) representing The target values
69 |
70 |         Returns
71 |         -----
72 |
73 |         X_new : numpy array of shape [n_samples, n_features_new] representing
the samples without outliers.
74 |         Y_new : numpy array of shape [n_samples] representing the labels
without outliers.
75 |
76 |         '''
77 |         decision = self.lof.fit_predict(X)
78 |         return X[(decision==1)],Y[(decision==1)]

```

Pseudo code pour le preprocessing (2) :

Nous avons inclus des tests unitaires

```

01 | if __name__=="__main__":
02 |     data_dir = 'public_data'
03 |     data_name = 'plankton'
04 |
05 |     Prepro = Preprocessor()
06 |
07 |     '''Show the original data before treatment'''
08 |     D = DataManager(data_name, data_dir) # Load data
09 |     print("*** Original data ***")
10 |     print(D)
11 |
12 |     '''Preproessing the data and show them after treatment'''
13 |     D.data['X_train'] = Prepro.fit_transform(D.data['X_train'], D.data['Y_train']
])
14 |     D.data['X_valid'] = Prepro.transform(D.data['X_valid'])
15 |     D.data['X_test'] = Prepro.transform(D.data['X_test'])
16 |     D.feat_name = np.array(['PC1', 'PC2'])
17 |     D.feat_type = np.array(['Numeric', 'Numeric'])
18 |     print("*** Transformed data ***")
19 |     print(D)
20 |
21 |     '''Show data after Outliers deletion'''
22 |     D.data['X_train'], D.data['Y_train'] = Prepro.outliersDeletion(D.data['
X_train'],D.data['Y_train'])
23 |     print("***Outliers Deletion***")
24 |     print(D)

```

Extrait de
model.py : https://github.com/adblackx/PLANKTON/blob/master/starting_kit/sample_code_submission/model.py **Pseudo code pour le model**

```
01 | def fit(self, X, y):
02 |     """
03 |         This is the training method: parameters are adjusted with
           training data.
04 |         In fact, we start by checking that the data are the same.
05 |         Then we check if THE HYPERPARAMETERS are already exists (by a
           precedent call of this function)
06 |         If is is the first time, then we use findModel in plkClassifier
07 |         findModel returns a voting model, it combines three models, with
           their best hyper parameters
08 |         find thanks to GridSearchCV, cross_val_score, "
           balanced_accuracy_score" use by plkAssitClassifier
09 |         Parameters
10 |         -----
11 |         X: Data
12 |         y: label (or classe)
13 |
14 |         This function check if
15 |         """
16 |
17 |         #checking the data are corrects
18 |         self.num_train_samples = X.shape[0]
19 |         if X.ndim>1:
20 |             self.num_feat = X.shape[1]
21 |
22 |         num_train_samples = y.shape[0]
23 |         if y.ndim>1:
24 |             self.num_labels = y.shape[1]
25 |         if (self.num_train_samples != num_train_samples):
26 |             print("model.py, fit: THERE IS A PROBLEM WITH THE DATA")
27 |
28 |         if not self.isFitted :#si on a pas deja entraine notre model, la
           premiere fois qu'on appelle le model, on commence ici par fit, on essaye de
           trouver les meilleurs parametres, on renvoie alors un voting
29 |
30 |         # on utilise le preprocessing ici
31 |         x1,y1 = prep.Preprocessor.construct_features(X,y)
32 |
33 |         class_to_find_voting_model = plkc.assistModel(x1,y1,prepo
           =prep.Preprocessor())
34 |
35 |         if self.dict_list == None: # on a la possibilité d'
           utiliser un dictionnaire pour gridSearch
36 |             d1 = {
37 |                 "name" : "ExtraTreesClassifier",
38 |                 "param_name" : ["n_estimators", "
           min_samples_split", "random_state"],
39 |                 "param_val" : [[200,250], [2], [2]]
40 |             }
41 |
42 |             d2 = {
43 |                 "name" : "RandomForestClassifier",
44 |                 "param_name" : ["n_estimators", "
           min_samples_split", "random_state"],
45 |                 "param_val" : [[200,250], [2], [2]]
46 |             }
47 |             self.dict_list = [d1, d2]
48 |             # cette methode classe trouve le meilleur model
49 |             #setBest= true veut dire qu'on a deja les models qui sont
           ionteressant, ils sont dans les listes qu'on lui donne
50 |             # si setBest= true on cherche les meilleurs models parmi la liste qu
           'on lui donne
51 |             class_to_find_voting_model.setModelsPrepro(self.
           model_name, self.model_list, self.dict_list, setBest=True)
```

```

52 |
53 |         #on obtiens le meilleur model de voting ici
54 |         voting_model = class_to_find_voting_model.getModelPrepro
55 |     ()
56 |         #utilisation de Pipeline, model + preprocessing donc
57 |         pipe_class = Pipeline([
58 |             ('preprocessing', self.prepo ),
59 |             ('voting', voting_model)
60 |         ])
61 |
62 |         self.clf = pipe_class
63 |         self.clf.fit(x1, y1)
64 |         self.isFitted = True # so we generate the best model here
65 |
66 |
67 |         else: # on rentre dans cette condition une fois qu'on a deja
68 |             # entraîné notre model, une fois qu'on deja notre model de voting
69 |             # it is the case that the best model is generated, so we
70 |             # load it, no need to fit again
71 |             # it takes too long time...
72 |             x1,y1 = prep.Preprocessor.construct_features(X,y) # on
73 |             reutilise le model ici
74 |             self.clf.fit(x1, y1) # on fit ici
75 |             print("FIT MON MODEL")

```

Pseudo code pour la visualisation :

```
01 | """
02 | Fonction pour les k-moyennes:
03 | X: ensemble d'apprentissage
04 | Y: labels
05 | param = paramètre des marker = [marker, size, color]
06 | color: couleur des points de chaque classe
07 | """
08 | def kmeans(X, Y, param, color, title, title_color, title_size, title_weight):
09 |     np.random.seed(42)
10 |     data = scale(X)
11 |     plot_colors = color
12 |     n_samples, n_features = data.shape
13 |     #Récupération du nombre de label
14 |     n_digits = len(np.unique(Y))
15 |
16 |     #Réduction en 2 dimensions des données en utilisant le PCA
17 |     reduced_data = PCA(n_components=2).fit_transform(data)
18 |     #Initialisation des k-moyennes
19 |     kmeans = KMeans(init='k-means++', n_clusters=n_digits, n_init=7)
20 |     #Entraînement des k-moyennes
21 |     kmeans.fit(reduced_data)
22 |
23 |     # Step size of the mesh. Decrease to increase the quality of the VQ.
24 |     h = .02      # point in the mesh [x_min, x_max]x[y_min, y_max].
25 |
26 |     # Plot the decision boundary. For that, we will assign a color to each
27 |     x_min, x_max = reduced_data[:, 0].min() - 1, reduced_data[:, 0].max() + 1
28 |     y_min, y_max = reduced_data[:, 1].min() - 1, reduced_data[:, 1].max() + 1
29 |     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
30 |
31 |     #Récupère le label de chaque donnée
32 |     Z = kmeans.predict(np.c_[xx.ravel(), yy.ravel()])
33 |
34 |     # Put the result into a color plot
35 |     Z = Z.reshape(xx.shape)
36 |     plt.figure(1)
37 |     plt.clf()
38 |     plt.imshow(Z, interpolation='nearest',
39 |               extent=(xx.min(), xx.max(), yy.min(), yy.max()),
40 |               cmap=plt.cm.Paired,
41 |               aspect='auto', origin='lower')
42 |     #Affichage des points
43 |     for i, color in zip(range(n_digits), plot_colors):
44 |         idx = np.where(y == i)
45 |         plt.plot(reduced_data[idx, 0], reduced_data[idx, 1], 'k.', markersize=2,
46 |                 color=color)
47 |     #Affichage des centres
48 |     centroids = kmeans.cluster_centers_
49 |     plt.scatter(centroids[:, 0], centroids[:, 1], marker=param[0], s=param[1],
50 |               linewidths=3, color=param[2], zorder=10)
51 |     #Affichage du titre
52 |     plt.title(title, color=title_color, fontsize = title_size, fontweight=
53 |             title_weight)
54 |     plt.xlim(x_min, x_max)
55 |     plt.ylim(y_min, y_max)
56 |     plt.xticks(())
57 |     plt.yticks(())
58 |     plt.show()
59 |
60 | """
61 | Fonction pour la surface de décision:
62 | X: ensemble d'apprentissage
63 | Y: labels
64 | nb_classes: nombre de classe
65 | classes_names: tableau contenant les noms des différentes classes
66 | Classifieur: classifieur utilisé
```

```

64 | """
65 | def decision_surface(X, Y, nb_classes, colors, classes_names, Classifieur, title,
    | title_color, title_size, title_weight):
66 |     #Nombre de classe
67 |     n_classes = 7
68 |
69 |     plot_colors = colors
70 |     plot_step = 0.02
71 |     data = scale(X)
72 |     #Récupération du nombre de label
73 |     n_digits = len(np.unique(Y))
74 |
75 |     #Réduction des données à 2 dimensions avec le PCA
76 |     X = PCA(n_components=2).fit_transform(data)
77 |
78 |     #Entraînement du classifieur
79 |     clf = Classifieur().fit(X, y)
80 |
81 |     #Affichage de la surface de décision
82 |     plt.subplot()
83 |
84 |     x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
85 |     y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
86 |     xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
    |                         np.arange(y_min, y_max, plot_step))
87 |
88 |
89 |     #Récupération des décisions prises par le classifieur
90 |     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
91 |     Z = Z.reshape(xx.shape)
92 |     cs = plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
93 |     plt.axis("tight")
94 |
95 |     #Affichage des points
96 |     for i, color in zip(range(n_classes), plot_colors):
97 |         idx = np.where(y == i)
98 |         plt.scatter(X[idx, 0], X[idx, 1], c=color, label=classes_names[i], s=3,
    |                 cmap=plt.cm.Paired)
99 |
100 |     #Affichage du titre
101 |     plt.axis("tight")
102 |     plt.suptitle(title, color=title_color, fontsize=title_size, fontweight=
    | title_weight)
103 |     plt.legend(title='Classes', bbox_to_anchor=(1.2, 0.5, 0.25, 0.), loc=5)
104 |     plt.show()
105 |
106 | """
107 | Fonction pour afficher le graphe des performances du modèle en fonction de la
    | taille de l'ensemble d'apprentissage avec des barres d'erreurs:
108 | t: tableau contenant les valeurs des abscisses
109 | res: valeur des ordonnées, correspondant au résultat de la fonction
    | model_performance
110 | p = paramètres = [curve_color, marker_type, marker_size, marker_color]
111 | """
112 | def plot_performance(t, res, p, title, title_color, title_size, title_weight):
113 |     #Affichage du graphe
114 |     plt.errorbar(t, res[0], res[1], color=p[0], marker=p[1], markersize=p[2],
    | MarkerFaceColor=p[3], MarkerEdgeColor="Black")
115 |     #Titre de l'axe des abscisses
116 |     plt.xlabel("Training set size")
117 |     #Titre de l'axe des ordonnées
118 |     plt.ylabel("Performance (score)")
119 |     #Affichage du titre
120 |     plt.title(title, color=title_color, fontsize=title_size, fontweight=
    | title_weight)

```

Références

- [1] The 5 Feature Selection Algorithms every Data Scientist should know,
<https://towardsdatascience.com/the-5-feature-selection-algorithms-every-data-scientist-need-to-know-3a6b566efd2>
- [2] Principal Component Analysis & Clustering with Airport Delay Data,
<https://gmacleenn.github.io/articles/airport-pca-analysis/>
- [3] Machine Learning : La mise à l'échelle (Feature Scaling),
<https://www.datacorner.fr/feature-scaling/>
- [4] Scikit-learn : PCA
<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>
- [5] Scikit-learn : SelectKBest
https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html
- [6] Scikit-learn : LocalOutlierFactor
<https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html>
- [7] Cours de L2 "Introduction à l'apprentissage automatique". Aurélien Decelle, 2019.
<https://www.lri.fr/~adecelle/site/pages/enseignements.html>
- [8] Cours de L2 "Mini-projet". Isabelle Guyon, 2020.
<https://sites.google.com/a/chalearn.org/saclay/home/info232-syllabus-2019-2020>
- [9] TP Mini-projet
<https://github.com/zhengying-liu/info232>
- [10] README.ipynb fourni avec le starting kit.
https://github.com/adblackx/PLANKTON/blob/master/starting_kit/README.ipynb
- [11] Scikit-learn : k-means
https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html#sphx-glr-auto-examples-cluster-plot-kmeans-digits-py
- [12] Scikit-learn : decision surface
https://scikit-learn.org/0.15/auto_examples/tree/plot_iris.html
- [13] Stacking : utilisation du model
<http://saclay.chalearn.org/home/info232-syllabus-2019-2020>
- [14] Scikit-learn Model : Schéma des différents modèles et de leurs utilisations selon les cas
https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html
- [15] Strategies pour la cross-validation
<https://www.kdnuggets.com/2017/08/dataiku-predictive-model-holdout-cross-validation.html>