IFT712 - Rapport de projet de session

Tom SARTORI Louis-Vincent CAPELLI Alexandre THEISSE

December 6, 2023

Contents

1 Contexte

Ce projet a été réalisé dans le cadre du cours **IFT712 - Techniques d'apprentissage** de l'université de Sherbrooke. Il consiste à utiliser des méthodes de classification de la bibliothèque *scikit-learn* sur une base de données Kaggle.

Le lien du GitHub associé est le suivant : `https://github.com/BigBaz54/leaf_classification`

2 Présentation de la base de données

Nous avons choisi la base de données *Leaf Classification* [1] comme conseillé dans le sujet du projet. Cette base de données contient des images de feuilles de différentes espèces. L'objectif est de prédire l'espèce d'une feuille à partir de ses caractéristiques (features) qui incluent 64 mesures représentant le contour de la feuille (margin) [16], 64 mesures représentant la forme de la feuille (shape) et 64 mesures représentant la texture de la feuille (texture).

La base de données contient 990 feuilles pour lesquelles l'espèce est connue et dont l'image est disponible. Les feuilles appartenant à 99 espèces différentes, il y a donc 10 feuilles par espèce.

3 Préparation des données

Afin de pouvoir utiliser les méthodes de classification de *scikit-learn*, nous avons dû préparer les données.

Les features ont été séparées des labels afin de pouvoir entraîner les modèles et ensuite évaluer leur performance en comparant les labels prédits avec les labels réels.

Nous avons également formé un ensemble d'entraînement et un ensemble de test à partir de l'ensemble d'entraînement initial fourni par Kaggle. Nous n'avons pas pu utiliser l'ensemble de test fourni par Kaggle car il ne contenait pas les labels correspondants aux échantillons.

Pour ce faire, nous avons utilisé la classe *StratifiedShuffleSplit* [14] de *scikit-learn* qui permet de séparer les données en conservant la proportion de chaque classe dans chaque ensemble.

4 Classification sur les features

Cette approche est celle attendue au vu de la base de données : elle consiste à prédire l'espèce d'une feuille à partir de ses caractéristiques. Nous avons donc utilisé les 192 features (margin, shape et texture) pour entraîner nos modèles.

4.1 Sélection des features

La première étape a été de sélectionner un sous-ensemble pertinent de features.

D'après notre compréhension de la base de données, elle contient 3 groupes de 64 features qui représentent respectivement le contour, la forme et la texture de la feuille. Nous avons donc considéré que chacun de ces 3 groupes devait être utilisé comme un tout représentant une seule et même information et que ce n'était pas pertinent de sélectionner des features individuellement au sein de ces groupes.

Également, nous avions l'intuition que ces 3 informations étaient complémentaires et qu'elles ne présentaient pas de redondance. Nous avons tout de même, pour s'en assurer, entraîné des modèles sur différents sous-ensembles des 3 groupes de features et comparé leurs performances. Ci-dessous, les résultats obtenus avec un SVC [15] :

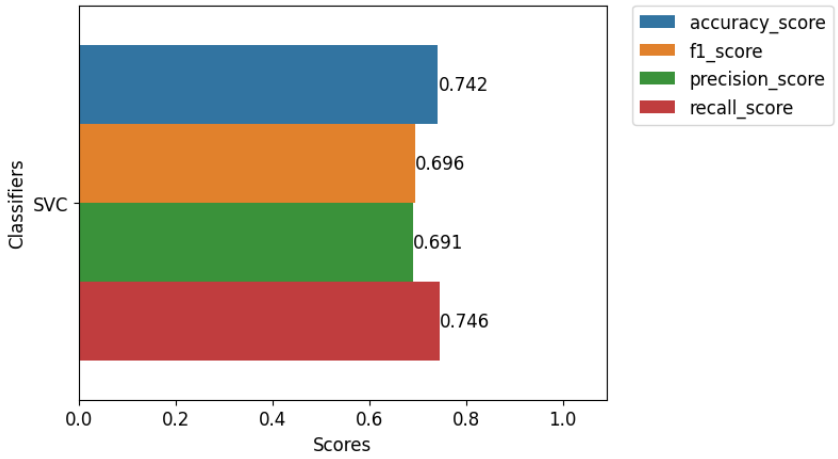

Figure 1: Performances d'un SVC sur les 128 features de shape et texture

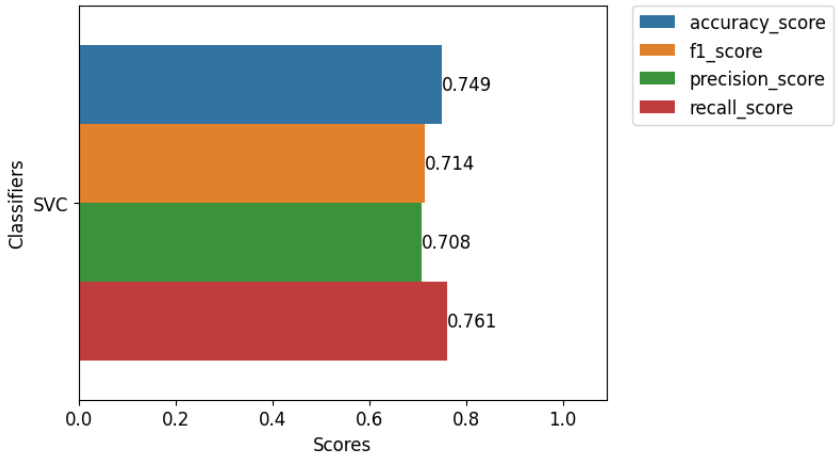

Figure 2: Performances d'un SVC sur les 128 features de margin et shape

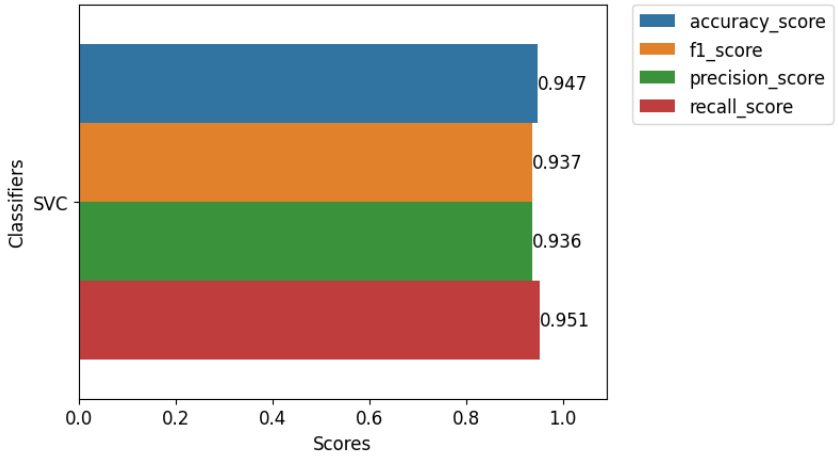

Figure 3: Performances d'un SVC sur l'ensemble des 192 features

Les résultats obtenus confirment notre intuition : les 3 groupes de features sont complémentaires et fournissent des meilleurs résultats lorsqu'ils sont utilisés ensemble. Nous avons donc décidé pour la suite de l'étude de considérer les 192 features à notre disposition.

4.2 Classification

4.2.1 Modèles utilisés

Nous avons utilisé les modèles suivants pour la classification :

- *DecisionTreeClassifier* [6] : arbre de décision
- *RandomForestClassifier* [12] : forêt aléatoire
- *BaggingClassifier* [4] : agrégation de modèles
- *LogisticRegression* [10] : régression logistique
- *SVC* [15] : machine à vecteurs de support
- *GaussianNB* [7] : modèle naïf bayésien gaussien
- *SGDClassifier* [13] : descente de gradient stochastique
- *KNeighborsClassifier* [9] : k plus proches voisins
- *GradientBoostingClassifier* [7] : gradient boosting
- *MLPClassifier* [11] : perceptron multicouche
- *AdaBoostClassifier* [3] : AdaBoost

Nous avons séléctionné ces modèles pour leur diversité, en effet ils utilisent des approches différentes pour la classification : certains sont des modèles linéaires, d'autres sont des modèles non-linéaires, certains sont des modèles à base de règles, d'autres sont des modèles à base de probabilités, etc.

Le but était de maximiser nos chances de trouver un modèle performant sur cette base de données en utilisant des modèles assez différents les uns des autres.

4.2.2 Cross-validation

Nous avons commencé par entraîner les modèles sur l'ensemble d'entraînement pour avoir un premier aperçu de leurs performances et une base de comparaison.

Voici les résultats obtenus :

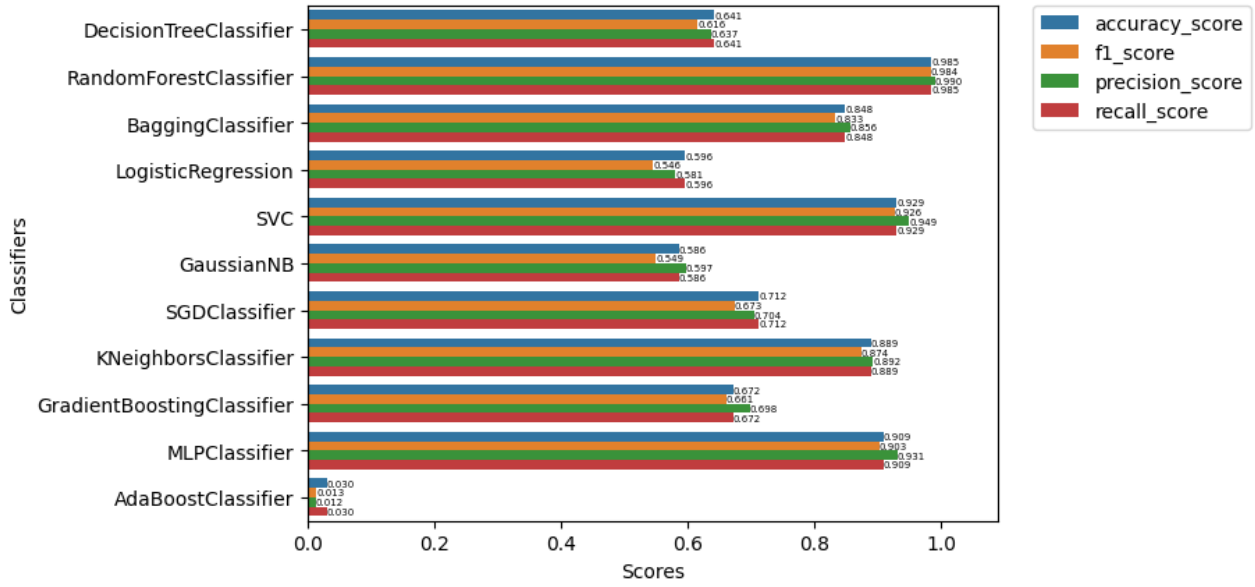

Figure 4: Performances des différents modèles sur tout l'ensemble d'entraînement

Nous avons ensuite utilisé la fonction *cross_validate* [5] de *scikit-learn* qui permet de faire une validation croisée avec un nombre de folds donné et d'utiliser le multi-threading pour accélérer le processus. Cette fonction utilise aussi la stratégie de *StratifiedShuffleSplit* [14] pour séparer les données en folds tout en conservant la proportion de chaque classe dans chaque fold.

Nous avons donc établi un nombre de folds maximal qui s'assure que chaque fold contient au moins un représentant de chaque classe.

Voici les résultats obtenus :

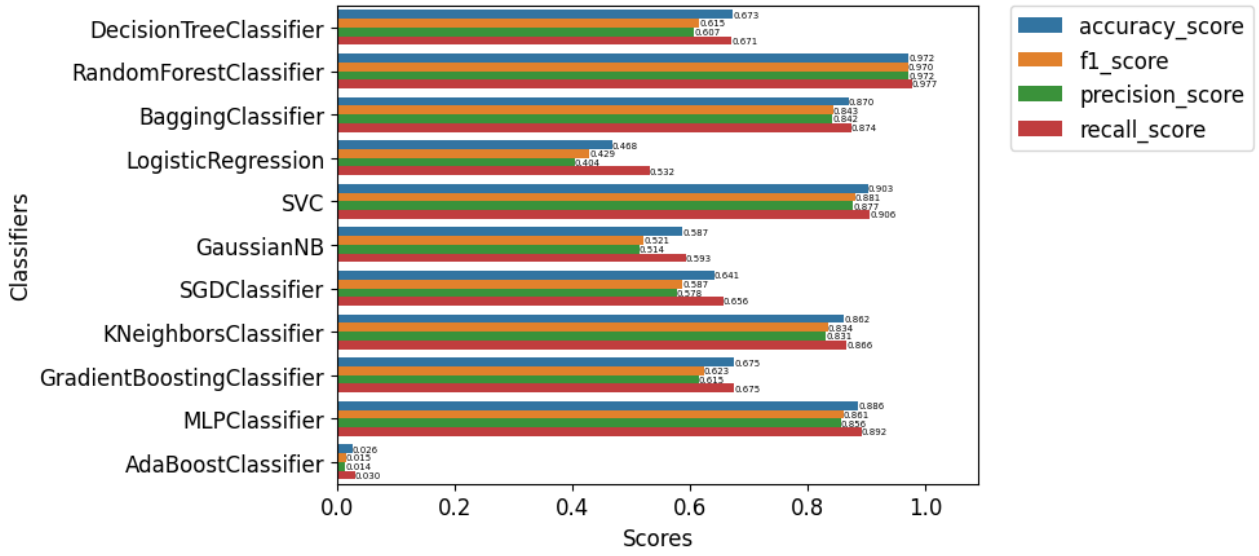

Figure 5: Performances des différents modèles avec une validation croisée à 10 folds

Les résultats obtenus sont similaires à ceux obtenus sans validation croisée, et sont même légèrement moins bons pour certains modèles. Cela peut s'expliquer par le fait que la base de données est assez petite (10 représentants par classe en tout, donc encore moins dans les données d'entraînement) et que la validation croisée réduit encore la taille de l'ensemble d'entraînement.

Dans ce cas, les modèles sont entraînés à plusieurs reprises sur des ensembles d'entraînement qui ne contiennent qu'un ou deux représentants de chaque classe, et ont donc plus de difficultés à généraliser. Cet effet est visible par exemple sur le graphique suivant, qui montre la précision obtenue pour les différents folds de la validation croisée d'un modèle de régression logistique :

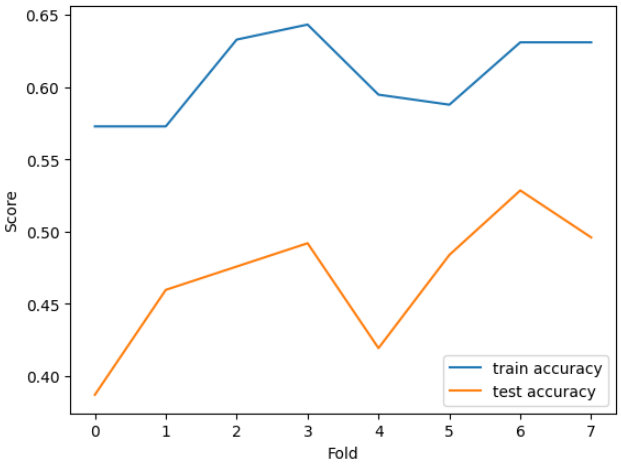

Figure 6: Performances d'un modèle de régression logistique sur les différents folds de la validation croisée

On constate également sur le graphique suivant que la précision de certains modèles (ici une forêt aléatoire) n'augmente pas linéairement avec le nombre de représentants de chaque classe dans l'ensemble d'entraînement et qu'il serait donc judicieux d'inclure au minimum 3 représentants de chaque classe dans chaque fold de la validation croisée :

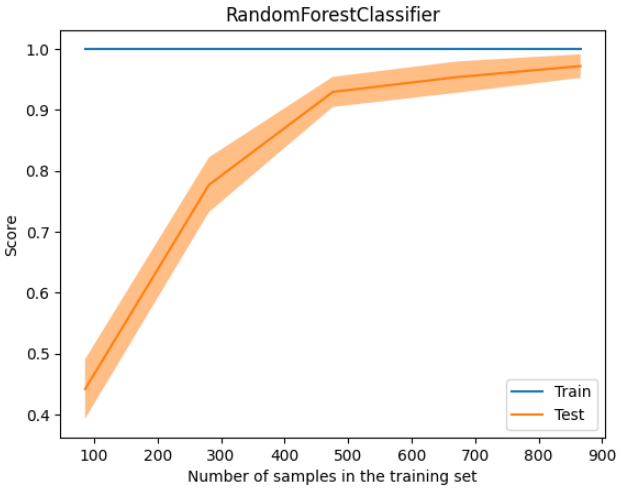

Figure 7: Performances d'un modèle de régression logistique sur les différents folds de la validation croisée

Nous avons donc décidé, pour réduire cet effet, de n'utiliser que 2 ou 3 folds pour la validation croisée pour la suite de l'étude.

4.2.3 Recherche d'hyperparamètres

Jusqu'à présent, nous avons utilisé les hyperparamètres par défaut des modèles. Pour améliorer leurs performances, il fallait donc les optimiser. Pour ce faire, nous avons utilisé la fonction *GridSearchCV* [8] de *scikit-learn* qui permet de faire une recherche d'hyperparamètres par validation croisée. La fonction *GridSearchCV* permet de spécifier une grille de valeurs pour chaque hyperparamètre et va tester toutes les combinaisons possibles pour trouver celle qui donne les meilleurs résultats. C'est une méthode d'exploration exhaustive qui est donc assez coûteuse en calcul.

Nous avons choisi les hyperparamètres à optimiser en fonction de la documentation des modèles et les valeurs à tester sont le fruit de plusieurs essais au cours desquels nous avons raffiné la grille de recherche pour s'assurer de ne pas passer à côté de valeurs intéressantes.

Les grilles de recherche utilisées sont disponibles dans le notebook *features_hyperparam_search.ipynb*, voici les valeurs optimales obtenues pour chaque modèle :

- *DecisionTreeClassifier* :
  - *criterion* : gini
  - *max_depth* : None
  - *max_features* : sqrt
  - *min_samples_leaf* : 1
  - *min_samples_split* : 5
  - *splitter* : best
- *RandomForestClassifier* :
  - *criterion* : gini
  - *max_features* : log2
  - *min_samples_split* : 2
  - *n_estimators* : 500
- *BaggingClassifier* :
  - *bootstrap* : False
  - *bootstrap_features* : True
  - *max_features* : 0.1
  - *max_samples* : 0.5
  - *n_estimators* : 100
- *SGDClassifier* :
  - *alpha* : 0.0001
  - *loss* : modified_huber
  - *max_iter* : 2000
  - *penalty* : l1
- *LogisticRegression* :
  - *C* : 1000
  - *max_iter* : 100
  - *penalty* : l2
  - *solver* : liblinear
- *GaussianNB* :
  - *var_smoothing* : 0.005
- *SVC* :
  - *C* : 50
  - *gamma* : scale
  - *kernel* : linear
- *KNeighborsClassifier* :
  - *algorithm* : auto
  - *leaf_size* : 1
  - *n_neighbors* : 1
  - *p* : 1
  - *weights* : distance
- *GradientBoostingClassifier* :
  - *learning_rate* : 0.025
  - *max_depth* : 2
  - *max_features* : log2
  - *min_samples_split* : 10
  - *n_estimators* : 500
- *MLPClassifier* :
  - *alpha* : 0.01
  - *hidden_layer_sizes* : (100,)
  - *learning_rate* : adaptive
  - *max_iter* : 500
  - *solver* : lbfgs
- *AdaBoostClassifier* :
  - *algorithm* : SAMME.R
  - *learning_rate* : 0.01
  - *n_estimators* : 500

4.3 Résultats

Avec les hyperparamètres optimisés, et en utilisant une validation croisée à 2 folds, voici les résultats obtenus :

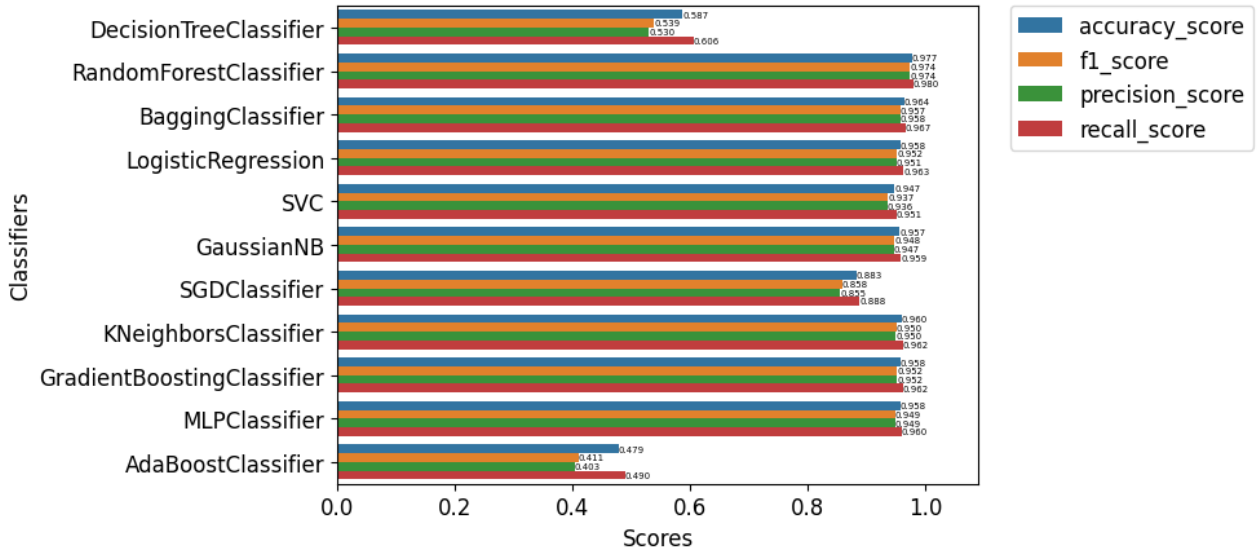

Figure 8: Performances des différents modèles avec une validation croisée à 2 folds et des hyperparamètres optimisés

Les performances sont bien meilleures qu'avec les hyperparamètres par défaut pour certains modèles, notamment LogisticRegression, GaussianNB, GradientBoostingClassifier et AdaBoostClassifier.

5 Classification sur les images

En plus de l'approche traditionnelle de classification sur les features, nous avons décidé d'essayer une approche de classification sur les images.

5.1 Prétraitement des images

Pour ce faire, nous avons utilisé la bibliothèque *Pandas* [2] pour transformer les images en tableaux de pixels.

Comme elles ne sont pas toutes de la même taille, nous avons dû les redimensionner pour que chacune soit représentée par une vecteur de même dimension. Nous avons essayé plusieurs manières, mais celle que nous avons retenue est de déterminer la taille maximale des images et de redimensionner toutes les images à cette taille en les centrant et en remplissant les pixels manquants avec des pixels noirs. Cette méthode permet de conserver la forme de la feuille et de ne pas les compresser afin de ne pas perdre d'information, au niveau des contours notamment.

Nous avons enfin applati ces tableaux pour obtenir finalement un vecteur de features de taille 1778337.

Les images étant en noir et blanc, parmi les 256 valeurs possibles pour chaque pixel, seules 2 sont utilisées : 0 pour le noir et 255 pour le blanc.

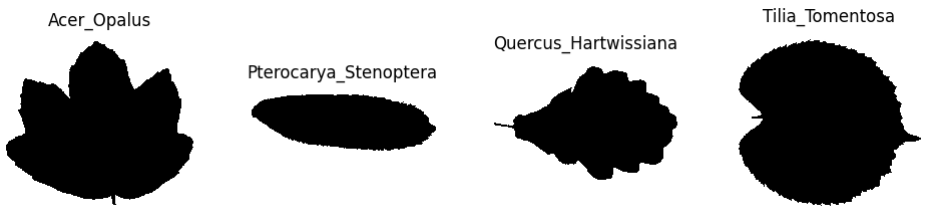

Figure 9: Exemples d'images de la base de données

L'approche que nous avons choisi pourrait poser problème car certaines feuilles se retrouvent plus petites que d'autres dans les images finales, mais nous avons estimé que cela ne devrait pas être le cas ici car même si les feuilles sont de tailles différentes, au sein d'une même classe elles ont généralement des tailles similaires.

Cet effet pourrait donc même être bénéfique pour la classification et pourrait permettre d'expliquer pourquoi le fait de redimensionner les images en les plaçant en haut à gauche (notre première approche) donne des résultats un peu moins bons que la méthode que nous avons finalement choisie.

5.2 Classification

5.2.1 Modèles utilisés

Nous avons utilisé les modèles suivants pour la classification basée sur les images :

- *RandomForestClassifier* [12]
- *KNeighborsClassifier* [9]

Ce sont les 2 modèles parmi ceux que nous avons utilisés pour la classification sur les features qui ont donné les meilleurs résultats et qui ont des temps d'entraînement assez faibles.

SVC, GradientBoostingClassifier et AdaBoostClassifier ont des temps d'exécution raisonnables mais ont donné de moins bons résultats donc nous ne les avons pas gardés car la recherche d'hyperparamètres était interminable.

5.2.2 Cross-validation

Nous avons utilisé la même stratégie de validation croisée que pour la classification sur les features, c'est-à-dire une validation croisée à 2 folds afin de ne pas réduire trop la taille de l'ensemble d'entraînement.

5.2.3 Recherche d'hyperparamètres

Nous avons utilisé la même stratégie de recherche d'hyperparamètres que pour la classification sur les features, c'est-à-dire une recherche exhaustive avec la fonction *GridSearchCV* [8] de *scikit-learn*. Les grilles de recherche utilisées sont disponibles dans le notebook *image_hyperparam_search.ipynb*, voici les valeurs optimales obtenues pour chaque modèle :

- *RandomForestClassifier* :
  - *criterion* : gini
  - *max_features* : log2
  - *min_samples_split* : 2
  - *n_estimators* : 500
- *KNeighborsClassifier* :
  - *algorithm* : TBD
  - *leaf_size* : TBD
  - *n_neighbors* : TBD
  - *p* : TBD
  - *weights* : TBD

5.3 Résultats

Voici les résultats obtenus sans et avec optimisation des hyperparamètres :

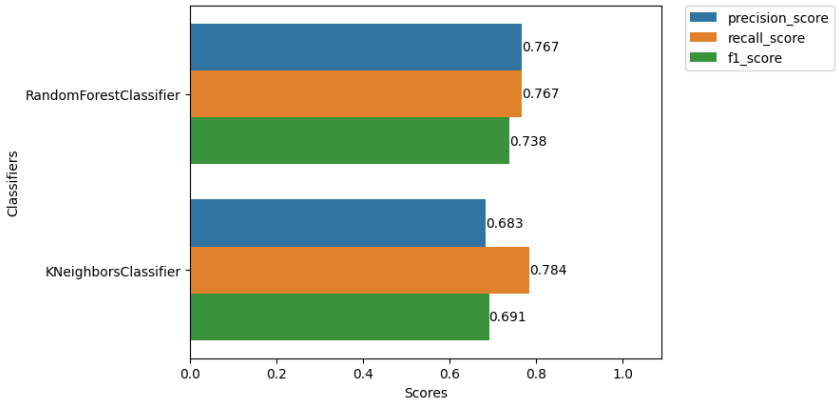

Figure 10: Performances des 2 modèles sans optimisation des hyperparamètres

Nous avons également créé des fonctions permettant de visualiser les prédictions des modèles sur les images de test. Voici quelques exemples :

Figure 11: Exemples de prédictions des 2 modèles sur les images de test

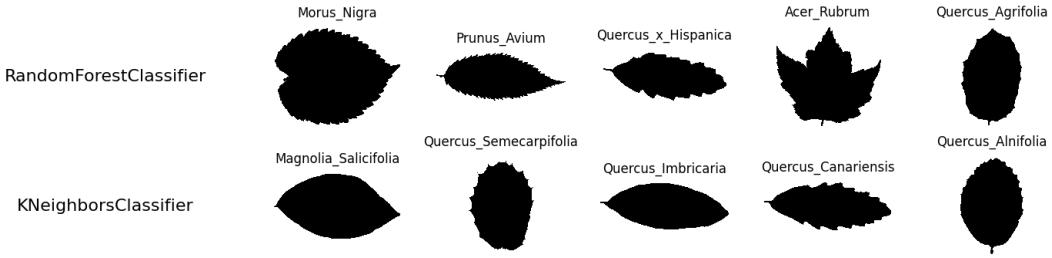

Figure 12: Classes présentant les moins bonnes performances

Nous sommes assez satisfaits des résultats obtenus, et même si les performances sont moins bonnes que celles obtenues avec la classification sur les features, nous avons trouvé intéressant de mettre en oeuvre cette approche alternative.

Les résultats moyens peuvent s'expliquer notamment par le nombre très bas de représentants de chaque classe, par l'hétérogénéité des tailles des images et par le fait que certaines images contiennent des feuilles dans des orientations différentes, ce qui rend la généralisation impossible.

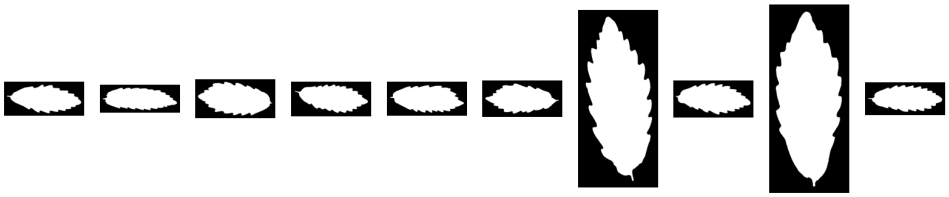

Figure 13: Exemples de classes avec des feuilles dans des orientations différentes

Également, la nature de la base de données a été un obstacle à la recherche d'hyperparamètres. En effet, la précision des contours des feuilles étant un critère important de différenciation des espèces, nous étions dans l'impossibilité de compresser les images pour réduire les temps de calcul et nous avons donc dû nous contenter de 2 modèles sur les 11 que nous avions utilisés pour la classification sur les features.

6 Conclusion

Nous sommes satisfaits des résultats obtenus, via les deux approches que nous avons utilisées et nous sommes fiers d'avoir fait preuve d'initiative en proposant une approche alternative de classification sur les images.

Ce projet nous a permis de mettre en pratique les connaissances acquises durant le cours afin de comprendre les algorithmes de classification et de choisir les paramètres à optimiser. Nous avons également pu mettre en pratique les notions de validation croisée et de recherche d'hyperparamètres et avons pu constater leur importance dans l'obtention de résultats satisfaisants.

References

[1] Will Cukierski Julia Elliott, Meghan O'Connell. Leaf classification. https://kaggle.com/competitions/leaf-classification, 2016.

[2] Pandas. Pandas. https://pandas.pydata.org/, 2023.

[3] Scikit-learn. Adaboost classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html, 2023.

[4] Scikit-learn. Bagging classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html, 2023.

[5] Scikit-learn. Cross validation. https://scikit-learn.org/stable/modules/cross_validation.html, 2023.

[6] Scikit-learn. Decision tree classifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html, 2023.

[7] Scikit-learn. Gaussian naive bayes classifier. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html, 2023.

[8] Scikit-learn. Grid search cross validation. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html, 2023.

[9] Scikit-learn. K-nearest neighbors classifier. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html, 2023.

[10] Scikit-learn. Logistic regression classifier. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html, 2023.

[11] Scikit-learn. Multi-layer perceptron classifier. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html, 2023.

[12] Scikit-learn. Random forest classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, 2023.

[13] Scikit-learn. Stochastic gradient descent classifier. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html, 2023.

[14] Scikit-learn. Stratified shuffle split. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html, 2023.

[15] Scikit-learn. Support vector classification. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html, 2023.

[16] Wikipedia. Leaf morphology. https://en.wikipedia.org/wiki/Glossary_of_leaf_morphology, 2023.