Equipe	Matricule
Joanny RABY	15062245
Dona CHADID	20102835

Rapport de projet
IFT 712 - Techniques d’apprentissage

UNIVERSITÉ DE SHERBROOKE

Sommaire

1. Introduction - Description des données

Pour le choix de base de données, il a été décidé d'utiliser la BD proposée : Leaf Classification du site Kaggle. Les fichiers fournis sont les suivants :

- **train.csv** - Ensemble des données étiquetées - d'une dimension : (990 exemples)
- **test.csv** - Ensemble de données non-étiquetées pour soumission Kaggle - d'une dimension : (594 exemples)
- **sample_submission.csv** - Un exemple de fichier de soumission pour Kaggle.
- **images** - les fichiers d'images (chaque image est nommée avec son identifiant correspondant).

Les données numériques furent choisies pour permettre de comparer les multiples classifieurs en se basant sur les mêmes entrées. Les fichiers trans.csv et test.csv contiennent les champs suivant:

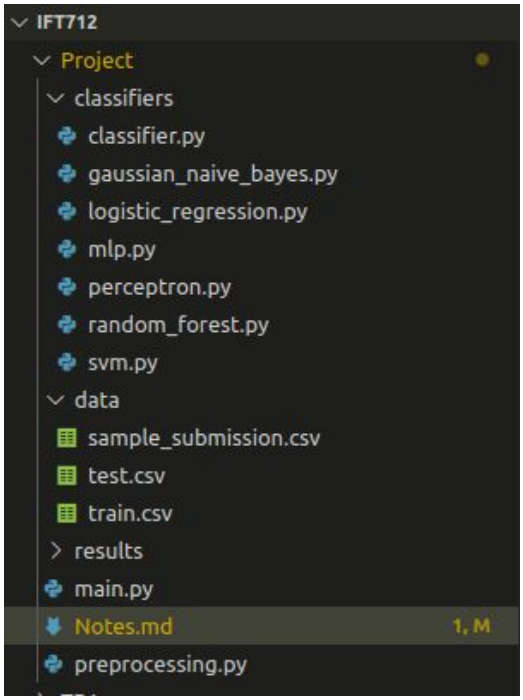
Champs de données

- **id** : id unique d'une image
- **species** : classe (unique à train.csv)
- **margin1, margin2, margin3, ..., margin64**
- **shape1, shape2, shape3, ..., shape64**
- **texture1, texture2, texture3, ..., texture64**

Les champs margin/shape/texture sont trois ensembles de caractéristiques extrait des images pour nous, et il y a 10 exemples pour chacune des 99 classes dans train.csv.

2. Choix de design

Il a été décidé de structurer le projet d'une manière simple et compréhensible. Un environnement virtuel respectant le *requirements.txt* donné pour les travaux pratiques (sur le Git ainsi que sur Moodle) est nécessaire.



Notre projet se compose de:

- **main.py :** Le fichier principal pour utiliser le code. Des démonstrations d'usage y sont présentées.
- **preprocessing.py :** Le fichier qui contient la classe de la gestion des données ("Preprocessor").
- **data :** Un dossier qui contient les données
- **results :** Un dossier contenant la sortie de différentes exécutions du main, ainsi que des résultats additionnels.
- **classifiers :** Le dossier que contient les classes des différents classifieurs utilisés, ainsi que leur classe parente "Classifier". Ci-dessous les fichiers contenus et les classes associées.
    - classifier.py : Classifier
    - gaussian_naive_bayes.py : NaiveBayes (Modèle génératif)
    - logistic_regression.py : LogisticRegression
    - mlp.py : MLP (Multilayer Perceptron)
    - perceptron.py : Perceptron
    - random_forest.py : RandomForest
    - svm.py : SVM

Les fichiers de code autre que **main.py** contiennent des *docstring* importants.

3. Démarche scientifique - Méthodologie

3.1 Traitement des données

Utilisation générale

C'est la classe Preprocessor qui s'occupe de faire passer les données de csv fournies par Kaggle à quelque chose d'utilisable par les différentes classifieurs. Par exemple, pour permettre un entraînement supervisé, ce qui est effectué par les six classifieurs choisis, les étapes suivantes sont effectuées, chacune avec une méthode de la classe:

- Importation initiale des données étiquetées (avec *import_labeled_data*)
- Encodage des étiquettes de classes en entiers (avec *encode_labels*)
- Séparation des données en ensembles d'entraînement et test (avec *train_test_split*)
- Normalisation des données (avec *scale_data*)
- Transformation par ACP si désiré (avec *apply_pca*)

La classe enregistre l'encodage des classes ainsi que les transformations effectuées pour utilisation future sur d'autres données, si nécessaire.

Séparation des données

Il est à noter que la séparation train/test est toujours effectuée de la même manière pour un même array d'entrée, car le mélange des données est fait avec un état aléatoire (*random state*) fixe. La séparation est faite en utilisant un algorithme k-bloc stratifié (StratifiedKFold), ce qui permet de conserver le même pourcentage de chaque classe entre les ensembles d'entraînement et test. Par défaut, la séparation est faite avec k=5, donnant des ensembles train/test de 80% et 20% des données respectivement. Puisque les données fournies contiennent 10 exemples pour chacune des 99 classes, on se retrouve alors avec 8 vs 2 exemples de chaque classe dans train vs test.

Normalisation des données

La normalisation des données étant généralement une pratique bénéfique, il a été décidé de systématiquement normaliser les données en utilisant le StandardScaler (soustraction de la moyenne et division par un écart-type pour chacune des 192 caractéristiques). Il ne semblait y avoir aucune contre-indication claire pour les classifieurs sélectionnés, et les résultats normalisés étaient prometteurs.

Analyse par composante principale (ACP/PCA)

L'ACP permet de réduire la dimensionnalité des données en transformant les caractéristiques en de nouvelles caractéristiques expliquant le plus de variance des données possible. (faible perte d'information). C'est utile lorsque l'entraînement est particulièrement long (avec LogisticRegression et MLP dans notre cas). Son utilisation est même nécessaire pour certains classifieurs (modèle génératif gaussien). Il est important de s'assurer de garder assez de composantes pour que les modèles aient l'information nécessaire pour distinguer les classes. Plus de détails sont disponibles dans l'analyse des résultats.

Prédire de nouvelles données

Pour prédire de nouvelles données sans étiquettes, l'importation des données via la méthode *import_unlabeled_data* est nécessaire, puis les transformations appliquées aux données d'entraînement doivent être appliquées à ces données. Il est également possible de désencoder (avec *invert_encoding*) les étiquettes numériques pour donner les classes originales.

3.2 Entraînement/validation des classifieurs

Validation croisée et optimisation d'hyperparamètres

Il a été décidé d'entraîner et valider par validation croisée et recherche par grille (avec GridSearchCV). En conséquence, chaque classifieur permet de fixer des hyperparamètres pour la validation croisée sur un seul ensemble d'hyperparamètres (avec *set_hyperparams*), ou bien des intervalles d'hyperparamètres (avec *set_hyperparams_range*).

Par défaut, la validation-croisée est effectuée avec k=8, ce qui crée des ensembles de validations où un seul exemple de chaque classe est présent (une extension de la logique *leave one out*). En effet, l'ensemble d'entraînement contient 8 exemples de chaque classe avec la séparation 80%/20% train/test initiale (celle faite par défaut).

Une fois la validation-croisée et la recherche d'hyperparamètre faite, le classifieur est ré-entraîné sur les données d'entraînement non séparées (donc 80% des données par défaut), avec les meilleurs hyperparamètres trouvés (ou ceux fixés). Ceci donne la forme "finale" d'un classifieur qui peut maintenant calculer des métriques sur l'ensemble de test ou de nouvelle données non-étiquetées (avec *predict*). Cette méthode est justifiée par les résultats observés : les métriques pour l'ensemble de test entrent dans un écart-type des métriques calculées pour les ensembles de validation (croisée). Ceci est discuté dans l'analyse des résultats.

Métriques de performance

Puisque les classes sont balancées (10 exemples pour chaque classe dans train.csv), il est attendu qu'utiliser la justesse (*accuracy*) comme métrique de performance de généralisation est correct. C'est la métrique qui est utilisée par défaut pour l'optimisation.
Comme gage de sûreté, le score F1 (moyenne harmonique de la précision et du rappel) est aussi calculé. Il est attendu que le score F1 macroscopique (moyenne des scores F1 pour chaque classe) soit légèrement inférieur à la justesse lorsque le classificateur ne développe pas une préférence pour certaines classes (erreurs concentrées sur certaines classes). Un score F1 plutôt bas par rapport à la justesse serait alors un indicateur potentiel de ce comportement. Si les classes n'étaient pas balancées, le score F1 devrait être favorisé pour l'optimisation, ce qui est possible lors de l'appel de l'entraînement.

Utilisation générale

La classe Classifier contient les méthodes principales utilisables par tous les classifieurs (entraînement, calcul de métriques, prédictions), et chaque classifieur individuel (qui hérite de Classifier) contient ses propres méthodes pour fixer ses hyperparamètres.

L'entraînement et l'estimation de la performance de généralisation prends la forme suivante en pratique:

- Transformer les données avec Preprocessor, selon les classifieurs choisis.
- Initialiser le (ou les) classifieur avec les données d'entraînement, qui incluent ici les données de validation.
- Fixer des hyperparamètres (avec *set_hyperparams* ou *set_hyperparams_range*). Des valeurs par défauts (présentées dans l'analyse des résultats) existent.
- Appeler l'entraînement/recherche d'hyperparamètre (avec *optimize_hyperparameters*). C'est à l'interne de ce processus que les séparations entraînement/validation sont faites.
- Affichage des meilleurs résultats (avec *display_general_validation_results*)
- Affichage des résultats exhaustifs de l'optimisation, si désiré (avec *display_cv_results)*
- Calcul de métriques sur l'ensemble de test (avec *get_accuracy* et *get_f1_score*)

Il est toujours possible d'enregistrer des objets Python spécifiques tel que l'instance du Preprocessor ou des classifieurs avec la librairie (standard) Pickle. Ceci s'avère pertinent si l'entraînement est de longue durée. L'utilisation d'un Notebook Jupyter est aussi possible.

3.3 Classifieurs et hyperparamètres optimisés

Les classifieurs choisis ont tous été présentés en classe, pour cette raison, une explication de base du fonctionnement n'est pas faite. Les hyperparamètres réglables et optimisés par recherche exhaustive (grille) sont présentés. Les résultats de ces optimisations sont présentes dans le dossier **results** du git, et sont analysés dans la prochaine section.

Perceptron

- *penalty* : Type de régularisation.
  Testé : Aucune, L1, L2, *elastic net* (L1+L2)
- *alpha* : Coefficient de régularisation.
  Testé : [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1]
- *eta0* : Taux d'apprentissage.
  Testé : [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

Régression logistique (Logistic Regression)

- *penalty* : Type de régularisation.
  Testé : *elastic net* (L1+L2)
- *C* : Inverse de la force de la régularisation.
  Testé : np.linspace(0.1, 10, num=10)
- *l1_ratio* : Fraction (entre 0 et 1) de C attribuée à la régularisation L1 dans *elastic net.*
  Testé : np.linspace(0, 1, num=10)

Le solveur "saga" est utilisé car c'est le seul supportant la régularisation *elastic net*.

Un grand C équivaut à peu ou pas de régularisation. De plus, la fraction de L1 permet de créer de la régularisation L1 ou L2 pure avec un seul paramètre (*l1_ratio*=1 et *l1_ratio*=0), contrairement aux hyperparamètres du Perceptron qui sont moins flexibles.

Ces hyperparamètres ont été testés sur des données modifiées par PCA (80% de variance expliquée = 23 composantes), le calcul étant long autrement.

Modèle génératif gaussien (Gaussian Naive Bayes)

Ce modèle n'a pas d'hyperparamètre direct à optimiser, le *var_smoothing* étant un paramètre pour la stabilité numérique qui ne semble pas influencer les résultats lors des tests préliminaires. Par contre, *var_smoothing* est tout de même implémenté comme le reste des hyperparamètres, pour unifier l'utilisation des différents classifieurs.

Pour ce classifieur, l'effet de la PCA avec différents nombres de composantes principales a été testé exhaustivement, avec ou sans *whitening (*voir documentation*)*.

Forêt aléatoire (Random forest)

- *criterion* : Critère d'impureté.
  Testé : gini, entropy
- *n_estimators* : Le nombre d'arbres de décision.
  - Testé : [50, 75, 100, 200, 300, 500]
- *max_depth* : La profondeur maximale d'un arbre de décision.
  Testé : [10, 25, 50, 75, 100, 200]

D'autres hyperparamètres sont ajustables, mais ils ont été ignorés puisque les résultats étaient déjà satisfaisants de cette façon.

Machine à vecteur de support (SVM)

- *kernel* : Noyau définissant la matrice de Gram (voir documentation).
  Testé : poly, sigmoid, rbf
- *C* : Paramètre de régularisation inversement proportionnel à la force de celle-ci
  Testé : [0.01, 0.1, 1, 2, 5, 10]
- *gamma* : Coefficient multiplicatif du kernel.
  Testé : np.geomspace(1e-3, 0.01, num=5), auto, scale
- *degree* (*poly* seulement) : Degré du polynôme.
  Testé : 1 à 5
- *coef0 (poly* et *sigmoid* seulement*)* : Coefficient additif du kernel.
  Testé : np.arange(0, 12, 2) pour *sigmoid*, np.arange(0, 8, 2) pour *poly*

Réseau neuronal simple (Multilayer Perceptron)

Un premier passage d'optimisation a été effectué avec des couches simples, puis un deuxième passage plus raffiné en fonction des résultats du premier.

- *hidden_layer_sizes* : Taille de la ou des couches cachées
  Testé : [100, (100, 20), (70, 10), (50, 30)] (4 configurations)
  puis [100, 150] (2 configurations)
- *activation* : Fonction d'activation des neurones
  Testé : relu
- *solver* : Algorithme d'optimisation des paramètres/poids.
  Testé : adam, lbfgs
- *alpha* : Force de la régularisation L2
  Testé : [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1] puis [0.1, 1, 5, 10]

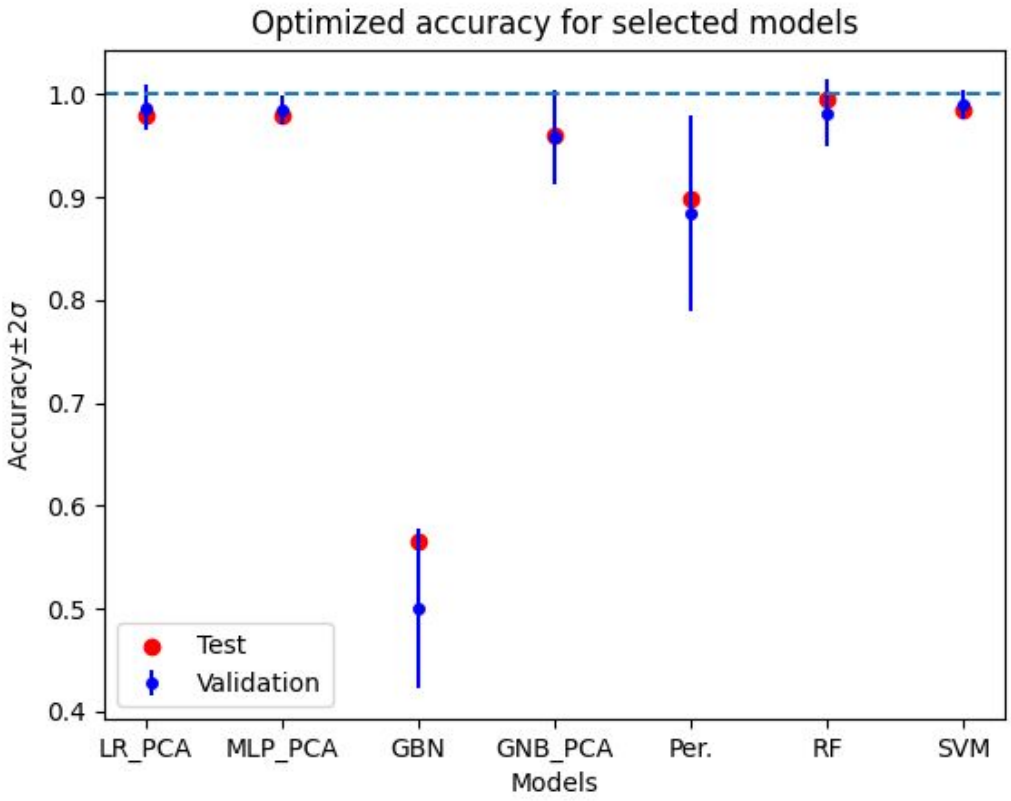
Ces hyperparamètres ont été testés sur des données modifiées par PCA (90% de variance expliquée = 44 composantes), le calcul étant long autrement. D'autres hyperparamètres non testés et non supportés par le code sont disponibles avec le solveur *sgd* (*stochastic gradient*

descent). Il a été supposé que les résultats par le solveur *adam* sont meilleurs qu'avec *sgd* (puisque celui-ci a été proposé comme une amélioration de *sgd*).

4. Analyse des résultats

Les résultats optimaux obtenus par la méthodologie expliquée et les intervalles d'hyperparamètres déjà décrits sont résumés dans *results/optimal_results.txt*. Ces résultats sont extraits des différents résultats de validation croisée disponibles dans les fichiers *results/*.out*. Les graphiques suivants résument les résultats optimaux. Les hyperparamètres optimaux ont été mis comme valeurs par défaut dans les fonctions "set_hyperparam" des différentes classifieurs.

4.1 Performance de généralisation



On rappel que les métriques de validation représentent des moyennes lorsque 70% des données sont utilisées pour l'entraînement et 10% pour la validation, pour 8 séparations différentes. Les métriques de test sont les valeurs obtenues lorsque l'entraînement est ré-effectué avec les paramètres optimisés (donc avec 80% des données totales au lieu de 70%).

À noter que l'intervalle de confiance sur les résultats de validation est considéré comme 95%, car les barres d'erreurs font 2 écarts-type.

On constate que la plupart des modèles performant très bien sur les ensembles de validation et de test, et que la valeur moyenne de la justesse obtenue par validation croisée est représentative de la capacité de généralisation des modèles. En effet, la justesse sur l'ensemble de test entre dans un écart-type de la justesse calculée par validation-croisée (sauf pour le modèle génératif non optimisé, GNB).

Le modèle génératif (GNB) performe plutôt mal (0.500±0.078) sans que les données soient passées par la PCA. Ceci est sans doute dû au fait que le modèle utilise comme prémisse que chaque paire de caractéristiques est indépendante. Une fois la PCA utilisée (GNB_PCA), les composantes principales, qui sont indépendantes, sont utilisées comme caractéristiques. On voit alors une hausse importante de la performance du modèle. La performance présentée est pour "whitening=False" et 27 composantes principales (0.958±0.046).

Le Perceptron (Per.) possède les pires résultats (0.885±0.095), ce qui semble indiquer que les données ne sont peut-être pas complètement séparables linéairement. Par contre, celui-ci converge en un nombre fini d'itération pour chaque ensemble de la validation-croisée, alors elles semblent l'être. Ainsi, autre chose est en cause. Comparons avec la régression logistique, (LR_PCA) qui est aussi un modèle linéaire, et performe significativement mieux (0.987±0.022). La cause principale de cette différence est probablement que la descente de gradient effectuée par le Perceptron mène à une frontière de décision pire que celle de la régression logistique. Lors d'une régression logistique, le gradient est calculé peu importe si un élément est bien classé ou non, contrairement au Perceptron. D'ailleurs, le grand écart type sur les résultats du Perceptron nous rappelle que la frontière de décision a seulement besoin de fonctionner pour que l'algorithme stoppe, et non que celle-ci soit bien "centrée" (comme avec le critère de Fisher). Ceci explique partiellement la grande variance de la justesse. Une cause secondaire est peut-être aussi que la régularisation *elastic net* avec un ratio variable de L1 et L2 n'est pas permise avec le Perceptron de sklearn. Le ratio est fixé à 0.5/0.5. Avec la régression logistique optimale, le ratio optimal trouvé est de 0.3 (conceptuellement, la force de la régularisation est multipliée par 30% pour le facteur L1, et 70% pour le facteur L2).

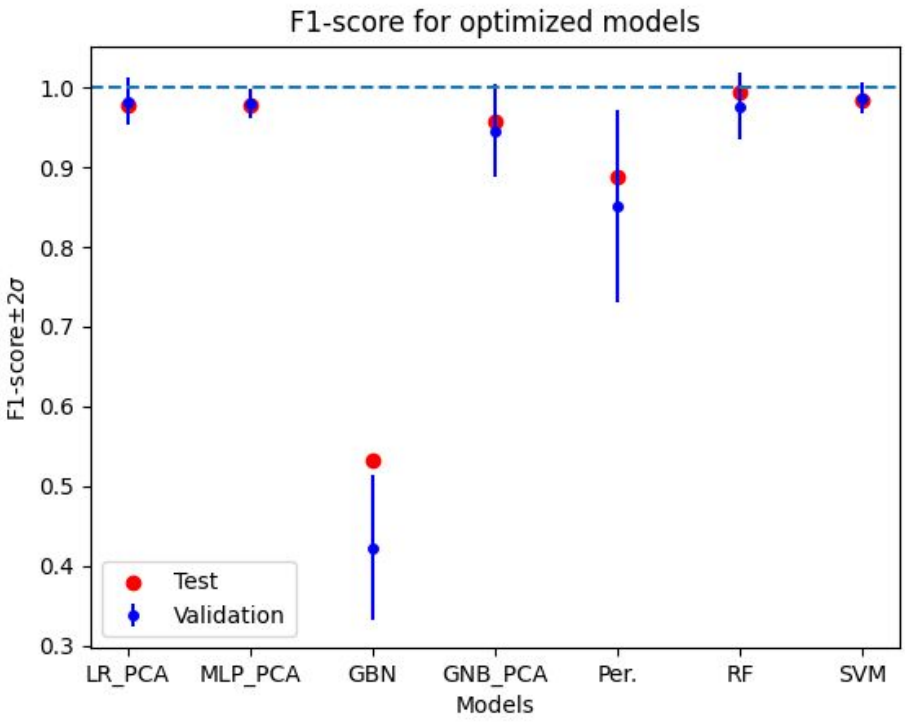
Comme mentionné, la régression logistique et le réseau neuronal simple (MLP_PCA) (0.985±0.014) utilisent de la PCA car le temps de calcul est long sinon. Avec plus de temps, une optimisation sur les données complètes aurait potentiellement pu donner une encore meilleure performance.

Il est à noter que la SVM n'a pas eu une variation très fine sur le paramètre gamma, alors sa performance est peut-être améliorable. Une légère modification de ce paramètre (0.00316 à 0.003) a été observée comme pouvant affecter significativement la performance.

Pour ce qui est de la SVM (0.990±0.014) et de la Forêt Aléatoire (RF) (0.982±0.032), PCA a aussi été testé (avec des composantes expliquant 80% de la variance), et elle donnait des résultats légèrement inférieurs. Potentiellement qu'une PCA avec plus de composante donnerait un résultat quasi-identique ou légèrement meilleur, mais les résultats étaient déjà excellents alors ceci ne fut pas davantage testé.

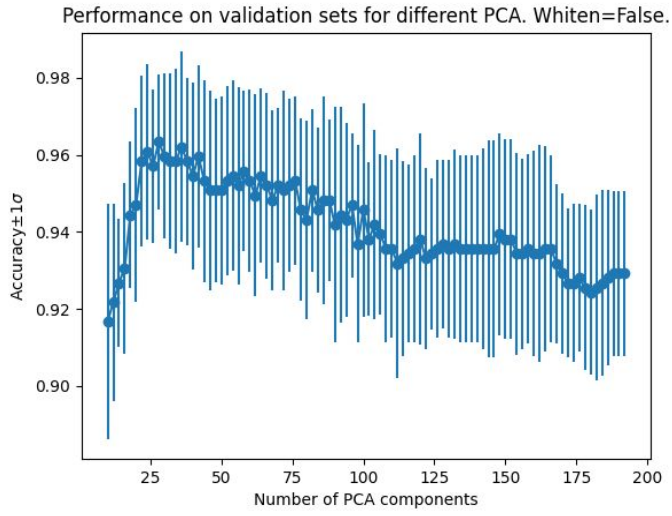
La Forêt Aléatoire présente les meilleurs résultats sur l'ensemble de test (justesse=0.995). Ceci semble être un coup de chance, car re-rouler l'entraînement même code donne parfois des valeurs différentes (justesse=0.985), avec des métriques de validation sont relativement stables. Ceci n'est pas étonnant, étant donné que c'est un classifieur qui a un grand élément de hasard, avec l'utilisation du bootstrapping.

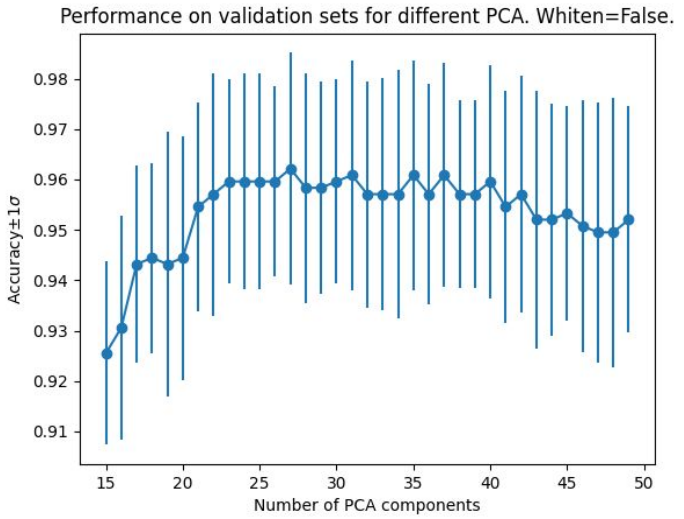
4.2 Scores F1

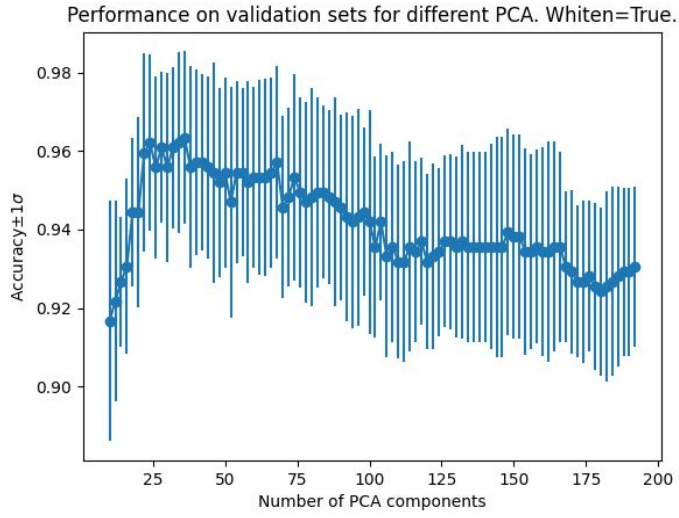


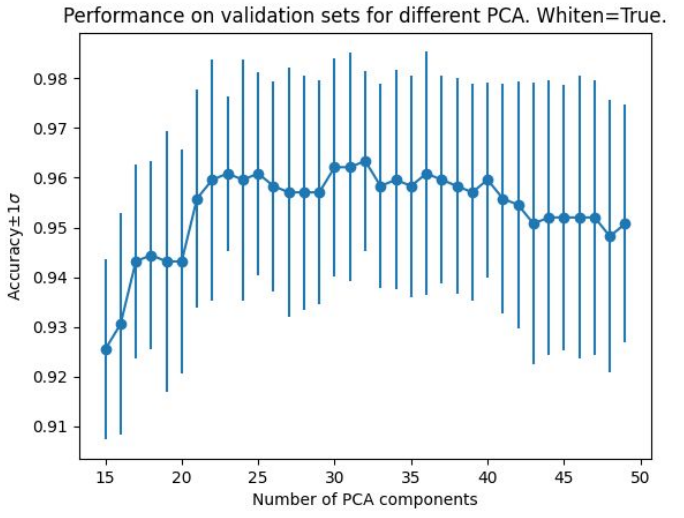
La figure précédente présente les scores F1 allant avec les justesses précédemment présentées. Les métriques viennent de la même optimisation de modèles. Comme attendu, le score F1 est légèrement inférieur à la justesse pour les modèles optimisés. Ceci signifie, comme expliqué précédemment, que le modèle ne fait pas des erreurs où il favorise fortement un classe (plusieurs classes différentes identifiées comme une seule classe). Ceci est excellent, et confirme qu'utiliser la justesse comme métrique de performance de généralisation est approprié. Ceci était attendu, car les classes sont balancées.

4.3 PCA

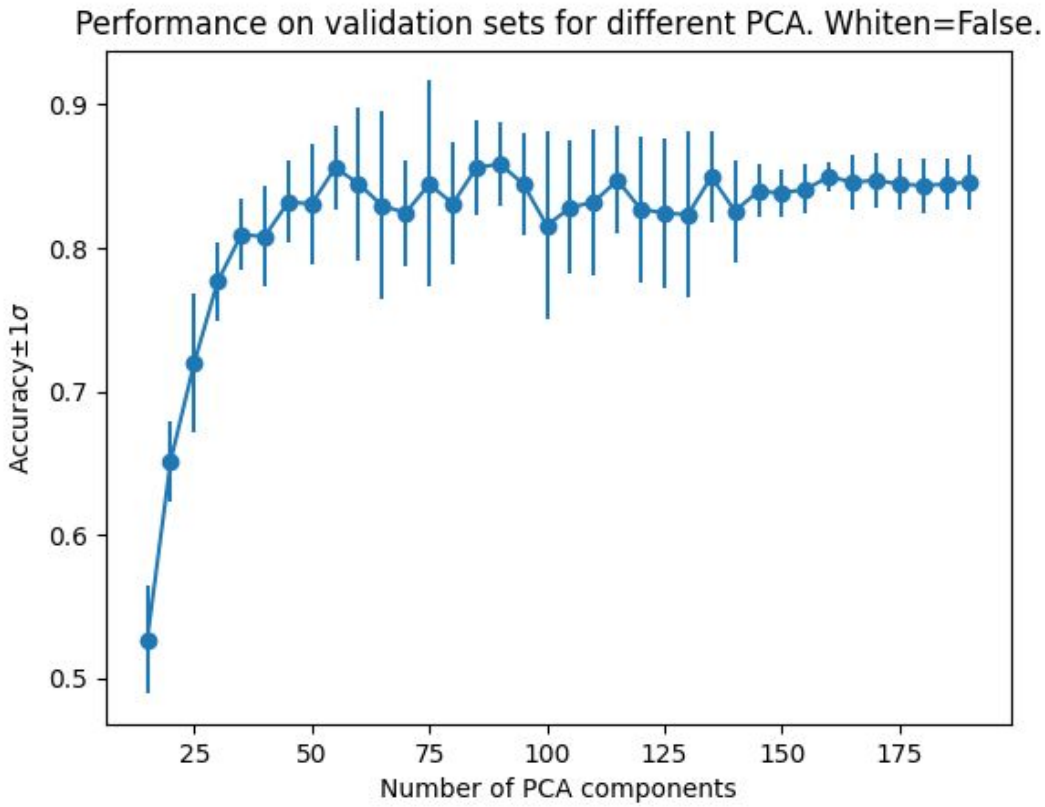








Les figures ci-dessus présentent l'effet de la PCA sur le modèle génératif. On constate que importe le nombre de composantes de PCA retenues lors de la transformation, pour 15 à 192 (nombre maximal) composantes, le résultat est tout de même meilleur qu'avec aucune PCA (justesse=0.500±0.078)). De plus, les résultats montrent qu'utiliser du *whitening* n'améliore pas les résultats.



Par curiosité, l'entraînement du Perceptron avec des données transformées par PCA a été testé avec les paramètres optimaux trouvés par la validation croisée initiale. (i.e. il n'y a pas eu de nouvelle optimisation d'hyperparamètres). Les résultats sont affichés ci-dessus. Le meilleur résultat, 0.859+-0.058, à 90 composantes, n'est pas meilleur que les résultats sans PCA, 0.885+/-0.095, ce qui indique que la transformation est néfaste ou neutre pour l'algorithme. La raison exacte n'a pas été déterminée.

5. Gestion de projet

Google Colab a été utilisé pour les tests très préliminaires du code ainsi que le développement de sa structure de base, ce qui mène à un commit initial sur Github plus gros que ce qui pourrait être attendu. Le développement subséquent a été entièrement effectué avec commits sur Github.

L'archive du code est aussi l'archive du code utilisé pour les autres travaux pratiques. Seul le dossier "Project" est pertinent. L'archive est "IFT712" hébergé sur le compte Github de Joanny Raby. La branche master est à jour avec les derniers changements.

6. Conclusion

Des classifieurs simples peuvent très bien performer pour classifier des feuilles d'arbre de la base de données Kaggle utilisée.

Au travers du rapport, il a déjà été mentionné pour quels modèles une recherche d'hyperparamètre plus poussée pourrait être facilement bénéfique (la plupart, probablement), mais certains aspects qui pourraient améliorer les résultats n'ont simplement pas été mentionnés. On y compte

- La vérification des données brutes pour éliminer les outliers.

- L'utilisation de normalisation différente.

- Application de la PCA sur les trois ensembles de caractéristiques, mais séparément. Peut-être serait-il bénéfique de ne pas appliquer la PCA sur certains ensembles de caractéristiques.

- Application des transformations (normalisation et PCA) intégrée à la validation-croisée. Techniquement, les transformations devraient être calculées avec l'ensemble d'entraînement seulement, puis appliquées aux ensembles de validation/test, sinon la performance peut-être surestimée. En ce moment, on utilise de l'information de l'ensemble de validation qui devrait être distinct pour les transformations, ce qui a un impact sur l'entraînement. Ainsi, les métriques pour la validation sont peut-être légèrement surestimées.

- L'utilisation du bootstrapping avec différents modèles.

- Prédiction en deux étapes : regroupement des données en genre (le genre est donné dans l'étiquette d'espèce) pour former des super-classes sur lesquelles des modèles sont entraînés, puis entraînement de modèles pour les espèces de chaque super-classe. On peut voir ceci comme une forme de prédiction aidée de normalisation par catégorie.

- Utilisation de métriques différentes pour l'optimisation des hyperparamètres.