IFT712 - Techniques d’apprentissage

Projet - Leaf

Salomé Gomez (19 156 930) - Julien Brochier (19 156 832)

Nous avons réalisé ce projet sur l'ensemble de données conseillé "leaf", qui consiste en de la classification d'espèces d'arbre à partir des caractéristiques de leurs feuilles.

Prétraitement des données

Nous avons commencé par parcourir nos données pour vérifier l**'absence de données manquantes**.
Nous avons ensuite étudié les données à la recherche de données catégoriques à convertir en données numériques, ce qui était le cas pour la catégorie 'species', que nous avons donc **labellisée**.

Nous avons séparé les données en 2 parties : les données d'entraînement et celles de test. Pour la séparation nous avons utilisé le ratio 80% test - 20% validation car c'est le standard en machine learning. Nous avons également séparé les données de leur cibles (catégorie 'species').

Enfin, nous avons réalisé une **normalisation** des données, ce qui permet d'avoir des données qui sont toutes à la même échelle. Un **ACP** (Analyse des Composantes Principales) a été réalisée ce qui permet de diminuer les dimensions des données en perdant pas ou peu d'information par le biais de la conversion de variables potentiellement corrélées en variables non corrélées. L'ACP permet un passage de 194 caractéristiques à 68.

Les différentes méthodes de classification implémentées

Critère de Fisher

Pour appliquer cette méthode, nous faisons l'hypothèse que les données sont caractérisées par une distribution gaussienne.
Le critère de Fisher est une stratégie utilisable sur 2 classes. Nous utilisons donc sa généralisation, l'**analyse discriminante linéaire (LDA)**, puisque nous avons 99 classes possibles. La LDA, consiste, comme le critère de Fisher en une projection des données multidimensionnelles, dans notre cas vers une dimension 98 D, sur laquelle les différentes classes sont séparées.
Comme la LDA se base sur une réduction des dimensions, ce qui est aussi le cas d'une ACP, nous nous sommes interrogés sur la validité de réaliser ces deux méthodes à la suite. Il n'est apparement pas contre indiqué de les utiliser de façon combinée. Nous avons néanmoins comparé les résultats avec et sans ACP (avec les paramètres par défaut de la classe), pour confirmer son apport :

Avec ACP	Sans ACP
Erreur train = 0.0012690355329949238 % Erreur test = 0.01485148514851485 %	Erreur train = 0.0 % Erreur test = 0.03608247422680412 %

L'absence de l'ACP mène à un potentiel sur-apprentissage et une erreur de test plus importante. Nous conservons donc l'ACP.
De ces premiers résultats, nous pouvons aussi faire l'hypothèse que nos données sont bel et bien linéairement séparables puisque nous obtenons des bons résultats en utilisant une méthode de classification linéaire.

La LDA se réalise avec la librairie sklearn.discriminant_analysis et sa classe **LinearDiscriminantAnalysis**.
Comme cette méthode est du type "closed form", il n'y a pas d'hyper paramètres à rechercher.
Néanmoins, on peut constater qu'il existe, dans la librairie, 3 méthodes de résolution :

- 'svd' : décomposition en valeurs singulières
- 'lsqr' : méthode des moindres carrés
- 'eigen' : decomposition en valeurs propres

Les deux dernières méthodes supportent le paramètre 'shrinkage' qui permet une régularisation du modèle, car la LDA est sensible au sur-apprentissage.

En lieu de la recherche d'hyper-paramètres, nous allons évaluer ces trois méthodes lors d'une validation croisée. Après avoir réalisé une 3-fold-cross-validation, nous obtenons les résultats suivants :

Solver sélectionné : eigen
Erreur train = 0.005082592121982211 %
Erreur test = 0.014778325123152709 %

La cross validation nous a ainsi permis de sélectionner le solver le plus adapté à nos données, celui correspondant à la décomposition en valeurs propres.
En vue de ces résultats, nous pouvons aussi affirmer que l'hypothèse de distribution gaussienne des données est appropriée.

Régression Logistique

Pour cette méthode, on utilise la classe **LogisticRegression** de la librairie sklearn.
Notre jeu de données étant petit il nous est possible d'utiliser le solver liblinear.

Après avoir calculé les erreurs d'entraînement et de test, on obtient les résultats suivants :

Erreur train = 0.0 %
Erreur test = 4.14522225031605564 %

On s’aperçoit alors que notre modèle fait de l’overfitting.
Une manière d’éviter l’overfitting est d’introduire un terme de régularisation. En prenant un terme de régularisation de 2.5, c’est à dire en fixant C = 0.4, C étant l’inverse du terme de régularisation dans la classe LogisticRegression, on obtient les résultats suivants :

Erreur train = 0.12391573729863693 %
Erreur test = 4.145 %

Cela a permis d’améliorer faiblement nos performances. Nous avons également essayé de diminuer le nombre d’itérations maximales mais cela n’a pas amélioré les performances. Pour nous en assurer, nous avons réalisé une cross-validation sur ces 2 paramètres.

La valeur du terme de régularisation obtenu varie légèrement entre deux exécutions du code mais elle est comprise entre 1 / 0.35 et 1 / 0.45.

```
C= 0.45  tol= 0.001
Erreur train =  0.12484394506866417 %
Erreur test =  1.0582010582010581 %
```

Arbre de décision

La méthode des arbres de décision consiste à créer des arbres dont les noeuds correspondent à différentes conditions sur les données et menant à des feuilles correspondant à nos classes.
Nous allons réaliser des arbres de **classification** grâce à la librairie sklearn.tree et la classe **DecisionTreeClassifier**.
Les arbres de décision sont très sensibles au sur-apprentissage, ce qu’on peut vérifier en lançant le modèles avec les paramètres par défaut de la classe DecisionTreeClassifier :

Erreur train = 0.0 %
Erreur test = 0.3217821782178218 %

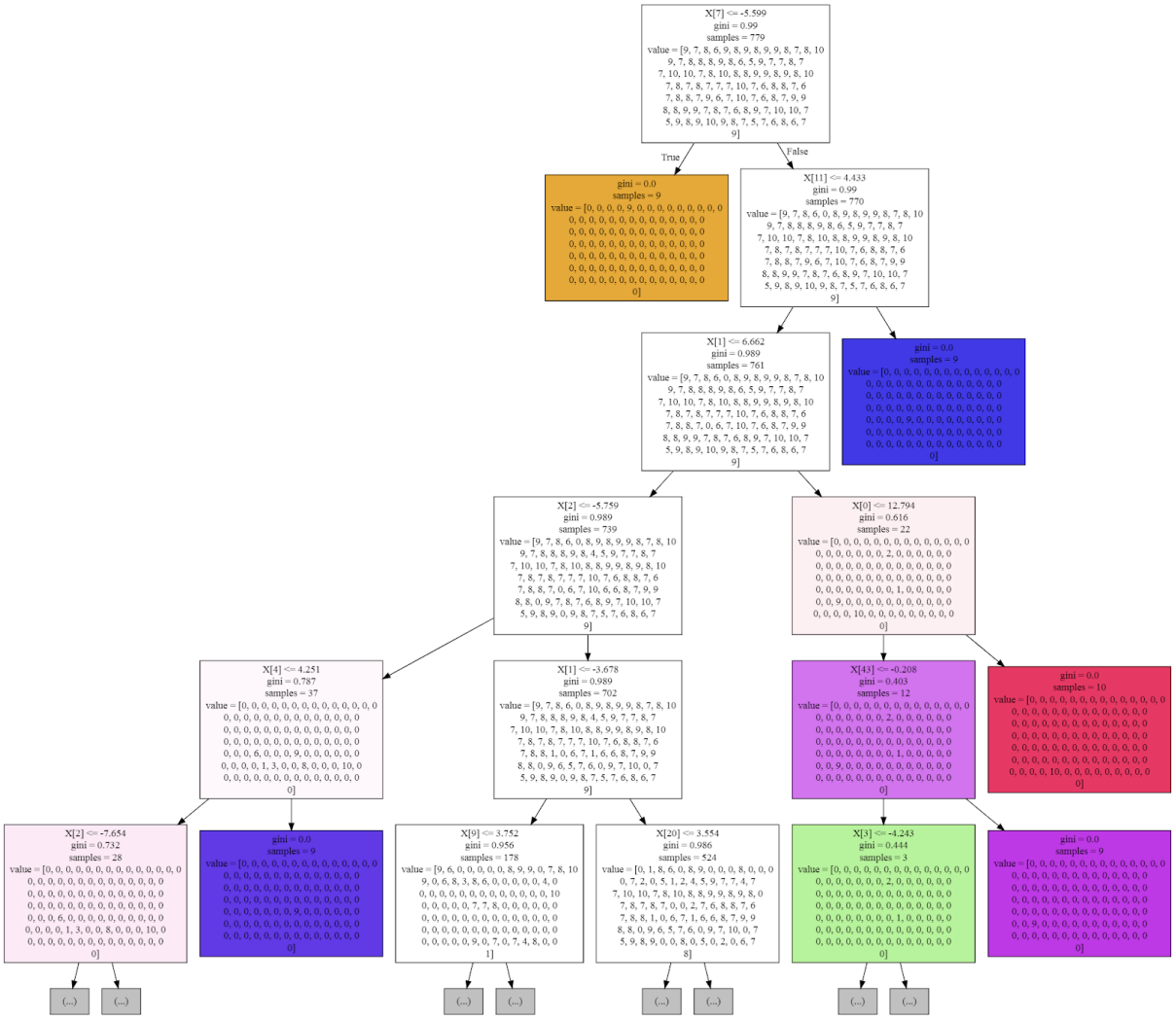
Pour éviter cela, il faut mettre en place des seuils d'arrêt, pour empêcher que l'arbre se développe trop. Cela se fait à travers de métriques d’impureté, qui permettent de décider si un noeud de l’arbre doit être subdivisé ou non.
Dans la classe utilisée, il existe deux mesures d’impureté : l’indice de Gini et l’entropie. Un paramètre associé à cette mesure est également ‘min_impurity_decrease’ qui détermine le seuil minimal pour la subdivision d’un noeud : si la subdivision entraîne une diminution de l'impureté supérieure au seuil elle est réalisée, sinon elle ne l’est pas.

Nous avons donc implémenté une 5-fold-cross validation pour sélectionner réaliser la meilleure sélection de ces paramètres en faisant varier ‘min_impurity_decrease’ entre 0 et 1 à un pas de 0.0001.
Nous obtenons les résultats suivants :

Mesure d'impureté : gini
min_impurity_decrease : 0.00030000000000000003
Erreur train = 0.25061425061425063 %
Erreur test = 0.5852272727272727 %

En les comparant avec les premiers résultats (obtenus avec les paramètres par défaut de la classe), nous pouvons remarquer que l'erreur de test augmente et que la différence entre l'erreur d'entraînement et l'erreur de test reste du même ordre. Il semble que nous dégradons nos résultats sans améliorer un potentiel problème de sur-apprentissage.

Nous allons donc conserver les paramètres par défaut de la classe, qui mènent à l'arbre suivant :



Machine à vecteur de support

Les SVM consistent à projeter les données dans un nouvel espace et à les séparer linéairement par un hyperplan en maximisant la marge.

Notre jeu de données étant petit, nous pouvons nous permettre d'utiliser des noyaux non-linéaires comme le rbf. Nous pensons que ce noyau conviendra mieux à notre situation car il est plus polyvalent.
Nous testons également les 3 autres principaux noyaux afin de nous en assurer.

Nous testons par cross validation nos paramètres C et gamma. Nous obtenons que la meilleure configuration est d'utiliser un SVM linéaire avec un coefficient de régularisation de 0.1.

C= 0.1 Kernel= linear gamma= auto
Erreur train = 0.0%
Erreur test = 0.966183574879227

Les différentes valeurs de C testées permettent de couvrir les différents types de SVM. Or, maintenant que nous avons déterminé que le SVM linéaire est préférable, nous avons voulu tester des valeurs de C plus restreintes autour de 0.1.
Nous avons testé 10 valeurs entre 0.01 et 1. Cependant cela ne nous a pas permis d'obtenir de meilleurs résultats, possiblement car en raffinant notre recherche de C nous commençons à sur-apprendre.

Réseau de neurones multicouche

Nous avons décidé de tester la méthode de réseau multicouche, plus précisément un **perceptron multicouche** permettant la classification. Pour ce faire, nous avons utilisé la classe **MLPClassifier** de la librairie sklearn.neural_network.
Les résultats obtenus avec les paramètres par défaut de la classe sont :

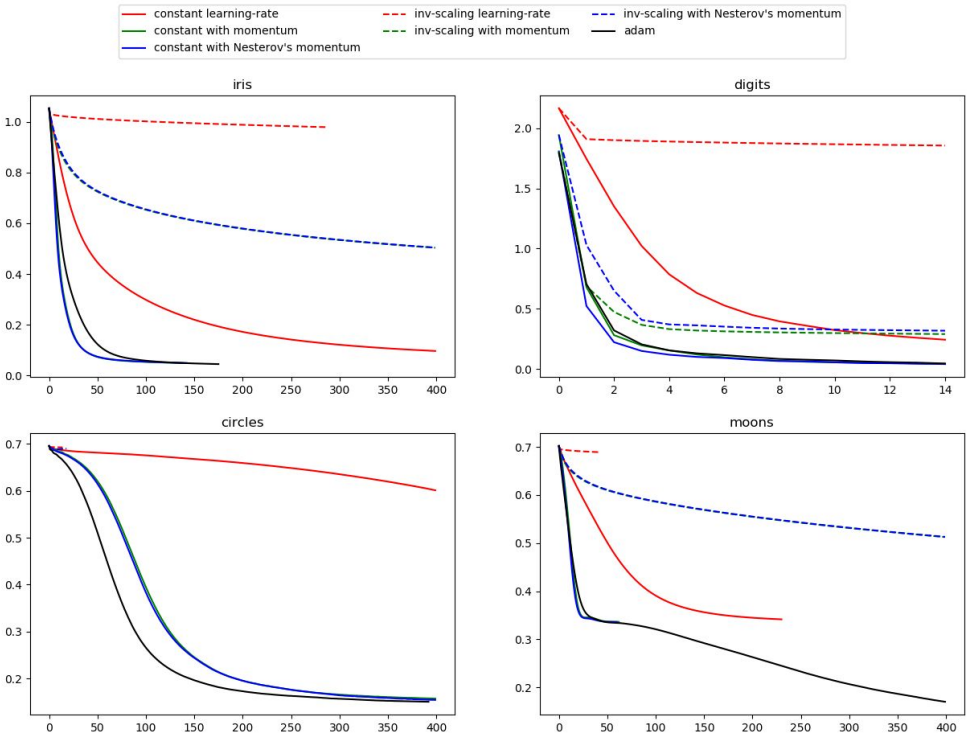
Erreur train = 0.0 %
Erreur test = 0.03553299492385787 %

On peut noter dans cette classe les différents paramètres ci dessous permettant d'influencer la performance du modèle :

- hidden_layer_sizes : qui correspond au nombre de neurones présents dans chacune des couches cachées. Cela permet d'influencer la capacité du modèle. Nous avons donc dû choisir différentes configurations de neurones.
  Au vu des résultats bons précédents obtenus grâce à des méthodes de classification linéaires (LDA, régression logistique), nos données semblent linéairement séparables et ainsi nous pourrions peut-être nous passer d'une couche cachée. Nous allons néanmoins essayer des configurations avec une ou deux couches cachées pour voir si cela améliore les performances. Une règle empirique pour la sélection du nombre de neurones dans les couches cachées est de choisir entre le nombre d'entrée et de sortie, c'est-à-dire entre 69 et 99. Nous allons choisir parmis ces données avec un pas de 5.

- activation : qui permet de sélectionner la fonction d'activation utilisée dans chaque couche. Nous décidons de tester les fonctions d'activation 'tanh' et 'relu' car elles nous paraissent être les plus intéressantes d'après le cours.

- solver : correspond à la méthode de calculs des différents poids w. Nous allons comparer les méthodes 'lbfgs' et 'adam'. Nous mettons de côté la méthode 'sgd' car dans la documentation, nous avons trouvé les courbes suivantes comparant la méthode 'sgd' avec 'adams':



  On peut constater que adams est la plupart du temps la meilleur méthode, ou a des résultats très similaires.

- alpha : qui correspond au coefficient de régularisation. Nous allons tester les valeurs suivantes : 0.0001,0.005 0.001,0.05, 0.01.

Nous avons obtenu les résultats suivants après une 3-fold-cross-validation sur ces paramètres, qui sont légèrement meilleurs que les résultats obtenus avec les paramètres par défaut :

Meilleurs paramètres trouvés :
{'activation': 'tanh', 'alpha': 0.005, 'hidden_layer_sizes': (99, 89), 'solver': 'adam'}
Loss associée :
0.9796954314720813
Erreur train = 0.0 %
Erreur test = 0.0297029702970297 %

Combinaison de modèles

Comme dernière méthode nous avons choisi de tester l'efficacité de la combinaison de nos modèles. Pour cela nous utilisons la classe **BaggingClassifier** de scikit-learn.
Nous avons opté pour le bagging car les modèles que nous utilisons actuellement ont une forte variance et ont tendance à overfitter.

Les résultats avec les paramètres par défaut sont :

Erreur train = 0.9975 %
Erreur test = 0.9631578947368421

On remarque qu'il existe un faible sous-apprentissage, mais il est nécessaire de comparer ces résultats avec la même méthode sans combinaison de modèles. En effet, en plus d'avoir créé une instance de la classe BaggingClassifier, nous utilisons la classe SVC, que nous avons aussi instanciée par défaut. Or dans ces conditions, un SVC présente nettement plus d'erreurs et du sur-apprentissage :

Erreur train = 0.0 %
Erreur test = 3.0303030303030303 %

En testant divers paramètres de bagging avec les paramètres de SVM qui avait apporté les meilleurs résultats, on obtient les résultats suivants :

Erreur train = 0.9962358845671268 %
Erreur test = 0.9948186528497409 %

Les résultats ne sont pas mieux que sans l'utilisation du bagging et nous sommes passé du sur apprentissage au sur-apprentissage. Par la suite, quelques soit les méthodes de classification utilisées, le bagging n'a pas permis de dépasser les performances obtenus avec la recherche des hyperparamètres par cross-validation.

Une autre idée de bagging que nous avions en tête était de **combiner** les résultats de nos **différents modèles**. Pour cela nous avons créé notre propre classe, qui effectue les prédictions en faisant un vote majoritaire de nos modèles. Comme notre classification comporte plus de deux classes, nous ne pouvions pas observer le signe de la somme des votes. On détermine donc la classe la plus votée à partir du mode des votes.

La classe **Custom_Combined_Models** peut être appelée avec ou sans cross validation. Dans le cas où elle ne recherche pas les meilleurs hyperparamètres, elle utilise ceux déterminés comme étant les meilleurs dans les sections précédentes.

Nous comparons ensuite son efficacité à celle des autres méthodes utilisées individuellement :

Erreur train = 0.12547051442910914 %
Erreur test = 2.5906735751295336 %

Erreur train LDA = 0.5018820577164366 %
Erreur test LDA = 2.5906735751295336 %
Erreur train LR = 0.12547051442910914 %
Erreur test LR = 2.5906735751295336 %
Erreur train SVC = 0.0 %
Erreur test SVC = 3.1088082901554404 %

Sur plusieurs essais on s'aperçoit que cette méthode ne permet pas toujours de dépasser la meilleure performance individuelle mais elle permet souvent d'égaler cette dernière. De même dans le cas de la cross validation.

Remarque :

Jusque là nous avions supposé que la séparation du jeu de données en m ensembles pour entraîner chaque modèle risquait de créer des ensembles trop petits, notre ensemble de départ étant formé de 990 données. Pour nous en assurer nous avons également testé ce cas.

Erreur train = 0.2560819462227913 %
Erreur test = 3.349282296650718 %

Erreur train LDA = 1.0243277848911652 %
Erreur test LDA = 6.220095693779904 %
Erreur train LR = 1.792573623559539 %
Erreur test LR = 5.263157894736842 %
Erreur train SVC = 0.5121638924455826 %
Erreur test SVC = 3.827751196172249 %

Nous constatons que dans notre cas, l'entraînement sur différents sous-ensembles n'apporte pas de changement significatif.

Retour sur la gestion de projet

En début de projet, nous nous sommes réuni pour choisir les méthodes à réaliser ainsi que leur répartition (3 méthodes chacun).
Nous avons également pris des décisions générales pour l'organisation du code :

- Chaque méthode est définie dans une classe
- le prétraitement des données et leur lancement se fait dans le fichier "main" grâce à une méthode "launcher"

- la métrique d’évaluation reste la même pour permettre de comparer les modèles et correspond au pourcentage de mauvaise classification

En terme d’organisation, nous avons utilisé trello pour avoir un suivi de l'évolution des tâches :
https://trello.com/invite/b/icBJlmUc/19eb4c8e54790aeafeba507dda68ee3c/ift-712-projet

Pour assurer une bonne gestion du code, nous avons utilisé github et avons crée des branches séparées pour chaque méthodes, que nous avons ensuite fusionné avec la branche principale lorsque les méthodes été implémentée et testées.