UNIVERSITÉ DE SHERBROOKE

UNIVERSITÉ DE SHERBROOKE
DÉPARTEMENT D'INFORMATIQUE
IFT 712 - Techniques d'apprentissage

Projet de techniques d'apprentissage

Classification de feuilles d'arbres

Travail présenté
à
Martin Vallières

Par
Hassen Chaker - 22 178 812

04 décembre 2022

Table des matières

1 Introduction

Le but de ce projet est de tester six méthodes de classification dans une base de données Kaggle. (www.kaggle.com) avec la bibliothèque Sklearn (scikit-learn.org). On a choisi d'utiliser la base de données sklearn recommandée, leaf-classification, parce qu'il présente l'avantage de proposer aussi bien des images que des attributs.

2 Prise en main des données et pré-traitements

2.1 Les données

Notre base de données est la base de données « leaf-classification » de Kaggle [1]. Elle contient 991 données étiquetées pour apprendre et 595 données de test non étiquetées. Chacune des 1584 feuilles se caractérise par 192 attributs et une image. J'ai essayé d'utiliser des images et des attributs pour savoir reconnaître une feuille. Et pour les attributs, ces derniers sont représentés dans un tableau de type csv facile à utiliser. Concernant les images, elles sont dans un dossier séparé et on va les intégrer par la suite.

2.2 Intégration des images

Afin d'intégrer les images, on a essayé dans un premier temps de normaliser leur taille. Comme ils sont tous de tailles différentes, ce qui fait qu'on a utilisé la bibliothèque Python scikit-image et en particulier la fonction resize pour le redimensionnement en taille 50*50.

FIGURE 2.1 – images de nos feuilles avant pré-traitement

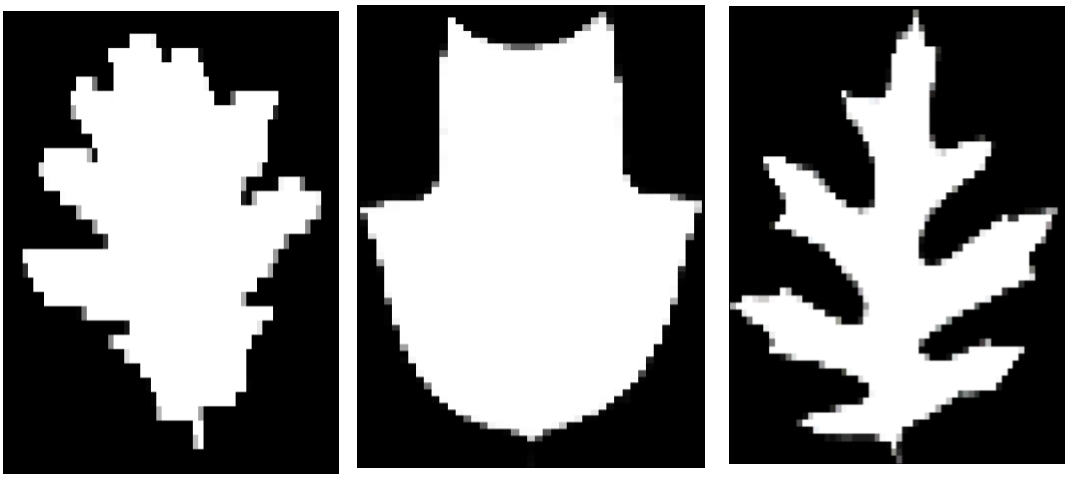

FIGURE 2.2 – images de nos feuilles après pré-traitement

Toutefois, il s'agit d'une amélioration purement visuelle. En effet, avec ou sans 0 padding, il n'existe pas d'amélioration réelle dans la classification. Après avoir réduit la taille des images, on peut les retrouver en consultant notre fichier et puis créer un tableau contenant une ligne de 2500 pixels pour chaque image. Alors on a créé un fichier csv d'entraînement de dimensions 990*2500 et on a aussi créé un autre fichier csv de test de dimensions 594*2500.Grâce à ce format, nous serons en mesure d'appliquer les méthodes de classification aux attributs de nos feuilles et aux images de la même façon.

2.3 Pré-traitement

J'ai séparé les données d'entraînement en deux parties ,80% des données sont consacrées pour l'entraînement et 20% des données pour le test. Pour la partie d'entraînement, j'ai supprimé la colonne "id" aussi que j'ai séparé la colonne "spécies». Le choix de la répartition des données est fait pour avoir des résultats de validation utiles et efficaces.

2.3.1 Vérification des données

2.3.1.1 Normalisation

Notre ensemble de données est complet et sont toutes comprises entre 0 et 1 donc on n'a pas besoin de la normalisation.

```
def normaliser(self):
    # normalise les données entre 0 et 1
    X = self.x_train # à supprimer
    #Utilisation de la méthode min Max pour normaliser les données
    scaler=preprocessing.MinMaxScaler()
    #Normalisation des valeurs
    d=scaler.fit_transform(X)
    #Retransformation en dataframe
    final_data=pd.DataFrame(d,columns=names)

    return(final_data)
```

FIGURE 2.3 – Quelques tests pour vérifier si les valeurs sont comprises entre 0 et 1

```
#Check la redondance des datas
print("Le nombre de données redondantes de train est " + str(df_train.duplicated().sum())+ "\n")
print("Le nombre de données redondantes de test est " + str(df_test.duplicated().sum())+ "\n")
#Affiche les attributs de train
print("Les attributs de train sont: " + "\n" + str(df_train.columns.values))
#Check si pour un des attributs il manque une valeur
print("Nb valeur manquante train " + str(df_train.isnull().any().sum()))
print("Nb valeur manquante " + str(df_test.isnull().any().sum()))
#A priori pas d'erreur dans les données
```

```
Le nombre de données redondantes de train est 0

Le nombre de données redondantes de test est 0

 Nb valeur manquante train 0
 Nb valeur manquante 0
```

FIGURE 2.4 – Quelques tests pour vérifier la redondance des données et si il y a des valeurs manquantes

2.3.2 PCA

Pour nos images nous avons 2500 colonnes de pixels. En effet, un pixel des coins est toujours noir alors qu'il existe un pixel au centre qui est toujours blanc. Ainsi, les colonnes correspondantes n'apportent aucune information d'où on va éliminer les colonnes de pixel inutiles .Puisque une variance nulle correspond à des pixels identiques on cherche à les éliminer en utilisant leur variance. Pour celles qui ont plus grand taux de variation, on va conserver toutes ses colonnes de pixel. Pour résumer, à travers la PCA et pour réduire le nombre de colonnes en fonction du ratio de variance nous avons codé une fonction.

```
#test prétraitement avec PCA
#on garde les colonnes  tel que la somme du ratio des variances soit supérieure au s
pretrait=Pretraitement(df_train)
pretrait.selection_avec_PCA(0.9)
pretrait=Pretraitement(df_train)
pretrait.selection_avec_PCA(0.95)
pretrait=Pretraitement(df_train)
pretrait.selection_avec_PCA(0.99)
```

```
pca variance ratio:  0.9045005231208898
nmb de colonne :  33
pca variance ratio:  0.9504847637817468
nmb de colonne :  49
pca variance ratio:  0.990500657703204
nmb de colonne :  89
```

FIGURE 2.5 – Évolution du nombre de colonnes sélectionnées en fonction de la quantité de la variance totale que nous souhaitons préserver

Pour classifier les images on a utilisé la méthode PCA qui est très bénéfique pour savoir le nombre de colonnes à réserver.

2.3.3 Les données aberrantes (outliers)

On a choisi le Z-Score comme un critère d'identification des outliers. Pour rappel le Z-Score [2] s'exprime comme cela :

$$Z = \frac{X - \mu}{\Sigma}$$

On a ainsi défini un seuil arbitraire (paramètre de la fonction de suppression d'outliers) tel que, si le Z-score le dépasse, on considère que la valeur est un outlier. On cherche alors les données qui possèdent au moins 10% de leurs attributs dont le Z-score est supérieur au seuil demandé et on les supprime de nos données.

```python
def supprimer_donnees_aberrantes(self,seuil:float,):
    # supprime les données aberante c'est à dire celle tel que: Z-score>seuil
    #Supprime les données dont 10% des attributs est considéré comme valeur aberrante

    #1 Calcul Z score, on le stocke
    z_scores=stats.zscore(self.x_train)
    #2 valeur absolue de ces valeurs, puisque un écart même négatif représente un outlier
    abs_z_scores = np.abs(z_scores)
    #3 Recupération des indices des outliers
    #dépend du seuil, ici on prend les données dont 10% des attributs sont considérés comme outlier
    Outliers_list=[]
    for i in range (abs_z_scores.shape[0]):
        #pour chaque attribut (192)
        compt=0
        for j in range (abs_z_scores.shape[1]):
            if (abs_z_scores.iloc[i,j]>seuil) :
                compt=compt+1
        if (compt>0.1*abs_z_scores.shape[1]):
            print(i)
            #print(abs_z_scores.loc[i])
            Outliers_list.append(i)
    #3 On ne garde que les données qui ne sont pas des outliers
    Filt=self.x_train[~self.x_train.index.isin(Outliers_list)]
    return(Filt)
```

FIGURE 2.6 – Données filtrées de leurs données aberrantes

Avec cette méthode on supprime les données aberrantes, c'est-à-dire c'est de supprimer les données dont 10% des attributs sont considéré comme valeur aberrante

3 Méthode de classifications

Tous les résultats présentés ci-dessous sont visible dans nos notebooks qui reprennent les différentes méthodes de façon structurée. Pour chaque résultat nous affichons la matrice de confusion et surtout la précision, le f1-score et le rappel de toutes nos classes qui présente une feuille mal classée. Nous affichons aussi la précision de chacun de nos essais et les meilleurs hyper- paramètres en cas de validation croisée. Au niveau du code, chaque méthode fait l'objet d'une classe tout comme les pré-traitements sur les attributs et sur les images.

3.1 Régression par la méthode du Perceptron

3.1.1 La méthode

La méthode de perceptron [3] est un classificateur linéaire qui utilise des données linéaires séparées. C'est le type de réseau de neurone le plus simple qui n'apporte pas une grande performance. On va utiliser dans notre cas entre deux classes une séparation linéaire pour ne pas avoir une confusion. Donc on opte pour le choix de séparation qui est basé sur cette méthode simple.

3.1.2 Choix des hyper-paramètres

Le choix est d'effectuer une variation sur le critère d'arrêt pour la perte (tol), le paramètre de régularisation (alpha), le nombre d'itérations maximal (max_iter) et sur la constante de multiplication des mises à jours (eta0).

Les meilleurs hyper-paramètres à utiliser sont :
- tol = 0.0001
- alpha = 0.0001
- max_iter = 400
- eta0 = 1

```
#pour enregistrer et faire varier nos hyper paramêtres
self.modele_perceptron = None
self.penalty = 'l2'
self.alpha = 0.0001
self.max_iter = 400
self.random_state = None  # essayer avec 1 aussi
self.tol = 0.0001
self.eta0 = 1
```

FIGURE 2.7 – Choix hyper-paramètres méthode de perceptron

3.1.3 Résultats

Dans un premier temps sans prétraitement et sans cross validation on obtient une précision de 12 %.

II) Classification par la méthode de perceptron :

a: méthode simple avec entrainement et prédiction sans cross validation ni prétraitement

```
#from classification_lineaire import Classification_Lineaire
x_train=df_train.drop(["species","id"], axis=1)
x_test=df_test.drop(["id"], axis=1)
x_etiquette=df_train["species"]
# application du modèle
model_perceptron=Classification_Lineaire(x_train,x_etiquette)
model_perceptron.entrainnement()
model_perceptron.prediction()
model_perceptron.resultats()
```

```
     Liriodendron_Tulipifera   0.000000  0.000000  0.000000         8
         Cytisus_Battandieri   0.044199  1.000000  0.084656         8
 Rhododendron_x_Russellianum   0.000000  0.000000  0.000000         8
                Alnus_Rubra    0.000000  0.000000  0.000000         8
      Eucalyptus_Glaucescens   0.000000  0.000000  0.000000         8
         Cercis_Siliquastrum   0.000000  0.000000  0.000000         8
           Cotinus_Coggygria   1.000000  0.125000  0.222222         8
           Celtis_Koraiensis   0.000000  0.000000  0.000000         8
         Quercus_Crassifolia   1.000000  0.125000  0.222222         8
```

```
precision =  0.125
matrice de confusion :
```

FIGURE 2.8 – Résultat sans prétraitement ni cross validation pour la méthode du Perceptron

Maintenant on effectue l'entraînement avec sélection d'attributs PCA mais sans cross validation :

```
# prétraitement des données avec PCA en conservant 90% de l'information selon le critère de la variance
pretrait=Pretraitement(df_train)
pretrait.selection_avec_PCA(0.90)
# application du modèle
model_perceptron=Classification_Lineaire(pretrait.x_train,x_etiquette, False)
model_perceptron.entrainnement()
model_perceptron.prediction()
model_perceptron.resultats()

# prétraitement des données avec PCA en conservant 95% de l'information selon le critère de la variance
pretrait=Pretraitement(df_train)
pretrait.selection_avec_PCA(0.95)
# application du modèle
model_perceptron=Classification_Lineaire(pretrait.x_train,x_etiquette, False)
model_perceptron.entrainnement()
model_perceptron.prediction()
model_perceptron.resultats()

# prétraitement des données avec PCA en conservant 99% de l'information selon le critère de la variance
pretrait=Pretraitement(df_train)
pretrait.selection_avec_PCA(0.99)
# application du modèle
model_perceptron=Classification_Lineaire(pretrait.x_train,x_etiquette, False)
model_perceptron.entrainnement()
model_perceptron.prediction()
model_perceptron.resultats()
```

```
precision =  0.29419191919191917
matrice de confusion :

precision =  0.23863636363636365
matrice de confusion :

precision =  0.23863636363636365
matrice de confusion :

precision =  0.16287878787878787
matrice de confusion :
```

FIGURE 2.9 – Résultat post PCA sans cross validation avec respectivement une variance de 90% 95% et 99% pour le Perceptron

Comme on peut le voir réduire le nombre de colonne avec la PCA est très bénéfique pour le modèle et réduit la complexité et assure une meilleure classification avec le perceptron.
Avec un cross validation :

perceptron avec cross validation

```
#from classification_lineaire import Classification_Lineaire
x_train=df_train.drop(["species","id"], axis=1)
x_test=df_test.drop(["id"], axis=1)
x_etiquette=df_train["species"]
# application du modèle
model_perceptron=Classification_Lineaire(x_train,x_etiquette, True)
model_perceptron.entrainnement()
model_perceptron.prediction()
model_perceptron.resultats()
```

```
{'alpha': 1e-05, 'eta0': 1, 'max_iter': 400, 'penalty': 'l2', 'tol': 1e-05}
                           precision    recall  f1-score   support

precision =  0.6515151515151515
matrice de confusion :
```

FIGURE 3.1 – Résultat avec Cross validation pour le Perceptron

On obtient une précision de 65%, donc on obtient un gain de 47% par rapport au paramètres originels, donc la cross-validation permet de nettement améliorer cette méthode.

3.2 Classification par les K plus proches voisins

3.2.1 La méthode

C'est une méthode des k plus proches voisins [4] qui consiste à classifier un objet en entrée selon le calcul de la distance qui renseigne sur l'écart entre les valeurs, ensuite de conserver les K valeurs qui sont les plus proches des précédents. Avec cette méthode on peut déterminer la classe majoritaire qui va par la suite nous permettre de classifier les données.

3.2.2 Choix des hyper-paramètres

Pour cette méthode le principe c'est de varier le maximum des paramètres afin de trouver les meilleurs valeurs donc on utilise la fonction de poids de la prédiction (weights) ,On a choisi de varier le nombre de plus proches voisins (num_neighbhors),le paramètre (leaf_size) qui a un effet sur la vitesse de calcul et un effet sur l'espace de

mémoire du calcul des arbres, la fonction de distance qui détermine les voisins les plus proches (metric) aussi que l'algorithme qui cherche les k plus proches voisins (algorithm).

Les meilleurs hyper-paramètres à utiliser sont :
• weights = 'uniform'
•num _neighbhors = 6
• leaf_size = 5
• metric = 'euclidean'
• algo = 'auto'

```
#pour stocker et modifier nos hyper-paramètres
self.num_neighbors = 6
self.weights = 'uniform'
self.metric = 'euclidean'
self.algo = 'auto'
self.leaf_size = 5
```

FIGURE 3.2 – Choix hyper-paramètres méthode des k plus proches voisins

3.2.3 Résultats

```
precision =  0.4797979797979798
matrice de confusion :
```

FIGURE 3.3 – Résultat sans cross validation ni prétraitement pour les K plus proches voisins

Entraînement avec sélection d'attributs PCA mais sans cross validation :

```
precision =  0.46464646464646464
matrice de confusion :
```

FIGURE 3.4 – Résultat post PCA sans cross validation avec respectivement une variance de 90% pour les K plus proches voisins

Maintenant on s'intéresse le plus avec un cross validation effectué sur les 3 PCA :

```
precision =  0.7234848484848485
matrice de confusion :
```

FIGURE 3.5 – Résultat post PCA à variance de 90% et cross validation pour les K plus proches voisins

On peut déduire que le cross validation permet l'amélioration de notre méthode.

```
precision =  0.7272727272727273
matrice de confusion :
```

FIGURE 3.6 – Résultat post PCA à variance de 95% et cross validation pour les K plus proches voisins

Même travail pour une variance de 95% pour la PCA.

```
precision =  0.7563131313131313
matrice de confusion :
```

FIGURE 3.7 – Résultat post PCA à variance de 99% et cross validation pour les K plus proches voisins

Pour finir on obtient une précision de 75.6% pour la PCA à variance de 99%, donc c'est un ensemble très performant et très meilleure que le modèle de Perceptron.
L'application de ces hyper paramètres pour l'ensemble de données de base nous a permis de gagner 37%.

```
precision =  0.8636363636363636
matrice de confusion :
```

FIGURE 3.8 – Résultat après cross validation pour les K plus proches voisins

Cette dernière précision montre la performance du. Nous avons ensuite essayé de l'appliquer à nos images.

```
precision =  0.3068181818181818
matrice de confusion :
```

FIGURE 3.9 – Résultat de la méthode K plus proches voisins

On finit donc avec un faible pourcentage de précision 30% qui présente un faible pourcentage pour la classification d'images.

3.3 Combinaison de modèles-Boosting (Decision Tree Classifier)

3.3.1 La méthode

La méthode de boosting [5] est basée sur combiner plusieurs classificateurs dans l'objectif d'atteindre une meilleure qualité de classification. Le principe issu de la combinaison de classificateurs c'est que les modèles qui ont les plus hautes performances apparaissent plus dans la décision finale de la classification aussi de booster la précision du modèle de base .C'est le but de l'utilisation de l'arbre de décision. L'arbre de décision représente un ensemble de choix qui sont interconnectés qui contient une probabilité pour chaque branche entre chaque nœud.

3.3.2 Choix des hyper-paramètres

Pour cette méthode le principe c'est de changer le nombre maximal d'estimateurs (n_estimators), la fonction qui produit des résultats différents dans chaque exécution (random_state) et le poids de chaque itération de la méthode de boosting (learning rate)

Les meilleurs hyper-paramètres à utiliser sont :
- n_estimators = 50
- learning_rate = 1.0
- random_state = None

```
#pour enregistrer et faire varier nos hyper paramêtres
self.n_estimators=50
self.learning_rate= 1.0 # paramètre d'apprenrtissage, faire varier de 0.25 à 1.5
self.random_state = None # essayer avec 1 aussi
```

FIGURE 3.10 – Choix hyper-paramètres méthode de boosting

3.3.3 Résultats

D'abord on obtient une précision de 36%.

```
precision =  0.3661616161616162
matrice de confusion :
```

FIGURE 3.11 – Résultat sans prétraitement ni cross validation pour la combinaison d'arbres de décision

On remarque que le modèle produit des résultats faibles donc on passe vers PCA et cross validation pour voir si on obtient de meilleurs résultats.

```
precision =  0.29797979797979796
precision =  0.273989898989899
precision =  0.2361111111111111
matrice de confusion :
```

FIGURE 3.12 – Résultat post PCA (pour respectivement 90%,95% et 99% de variance) mais sans cross validation pour la combinaison d'arbres de décision

C'est encore moins précis que celui le précèdent, il existe moins de possibilités pour les arbres de décision.
Le résultat pour le post cross validation.

```
precision =  0.24494949494949494
matrice de confusion :

precision =  0.3661616161616162
matrice de confusion :
```

FIGURE 3.13 – Résultat après cross validation (pour respectivement l'ensemble après PCA à 99% et l'ensemble de base) pour la combinaison d'arbres de décision

Il existe un mauvais choix d'hyper paramètres.

3.4 Les machines à vecteurs de support

3.4.1 La méthode

La méthode des machines à vecteurs de support [6] est performante pour les jeux de données et spécialement dans le cas du faible nombre d'échantillons qu'on possède dans notre projet. Ils ont rapidement été adoptés pour leur capacité à travailler avec des données de grandes dimensions, le faible nombre de hyper-paramètres, leurs garanties théoriques, et leurs bons résultats en pratique.

3.4.2 Choix des hyper-paramètres

Pour cette méthode le principe c'est de fixer les le paramètre en mode auto (gamma) aussi que le paramètre de régularisation (C)

. gamma = 'scale'
. C = 1.0

```
self.C= 1.0 # paramètre de régularisation, faire varier de 0.05 à 1.5
self.gamma='scale' #scale ou auto
```

FIGURE 3.14 – Choix hyper-paramètres méthode de machines a vecteurs de support

3.4.3 Résultats

On obtient sans le PCA une précision de 72%.

```
precision =  0.726010101010101
```

FIGURE 3.15 – Résultat pour les machines à vecteurs de support sans prétraitement ni PCA ou cross validation

On obtient un résultat très performant :

```
precision =  0.7032828282828283
precision =  0.7234848484848485
precision =  0.7323232323232324
```

FIGURE 3.16 – Résultat pour les machines à vecteurs de support après PCA (à respectivement 90%, 95% et 99% de variance) sans cross validation

On peut déduire qu'il n'est pas nécessaire de conserver toutes les colonnes pour cette méthode.

```
precision =  0.696969696969697
```

FIGURE 3.17 – Résultat pour les machines à vecteurs de support après cross validation

On obtient un résultat qui n'est pas bonnes. Donc on peut déduire que les hyper paramètres ne sont pas optimaux pour notre ensemble de départ et ce qui explique une précision diminuée.

3.5 Les réseaux de neurones

3.5.1 La méthode

Un réseau neuronal [7] est une méthode plus ou moins complexe. Sa méthode se distingue par son architecture, son niveau de complexité (le nombre de neurones, présence ou non de boucles de rétroaction dans le réseau), par le type des neurones (leurs fonctions de transition ou d'activation).

3.5.2 Choix des hyper-paramètres

Pour cette méthode, on va définir le nombre de couches et le nombre de neurones.
Le principe c'est de tester le nombre d'itérations maximal (max_iter), de changer le nombre de sous-couches (hidden_layer_sizes) et de tester les validations croisées.
L'objectif est de varier le nombre d'itérations maximal (max_iter) pour savoir l'emplacement du palier d'apprentissage.

Voici les résultats de nos validations croisées :

```
self.modele_reseaux_neuronnes = Reseaux_Neuronnes(self.x_train, self.y_train, False, self.x_test, True)
self.modele_reseaux_neuronnes.hidden_layer_sizes=(200,)
self.modele_reseaux_neuronnes.max_iter=800
self.modele_reseaux_neuronnes.random_state=None
self.pond_reseaux_neuronnes = 0.81
```

FIGURE 3.18 – Choix hyper-paramètres méthode de réseaux de neurones

Avec une seule couche cachée de 200 neurones on obtient des résultats qui sont meilleurs et avec beaucoup d'itérations, donc on va utiliser ces paramètres dans ce modèle.

3.5.3 Résultats

Résultat pour les réseaux de neurones sans prétraitement ni cross validation et avec une précision importante de 65%.

```
precision =  0.6565656565656566
```

FIGURE 3.19 – Résultat pour les réseaux de neurones sans prétraitement ni cross validation

On a appliqué les hyper paramètres qu'on a trouvé avec notre cross validation et on obtient une précision de 82%.

```
precision =  0.8232323232323232
```

FIGURE 3.20– Résultat pour les réseaux de neurones après cross validation

Avec ce résultat on peut déduire que c'est une partie des meilleures méthodes de classification puisqu'elle produit une précision de 82%.

3.6 Les forêts aléatoires

3.6.1 La méthode

L'algorithme des forêts d'arbres décisionnels [8] effectue un apprentissage sur de multiples arbres de décision entraînés sur des sous-ensembles de données légèrement différents.

3.6.2 Choix des hyper-paramètres

Pour cette méthode le principe c'est de définir le nombre minimal de données sur un nœud (min_samples_split), le nombre d'arbres générés (nmb_arbre) et le critère de décision (criterion).

. min_samples_split = 2
. nmb_arbre = 200
. criterion = gini

```
#pour stocker et modifier les hyper-paramètres
self.modele_random_forest = None
self.nmb_arbre = 200 # faire varier de 100 à 500
self.criterion = 'gini' # gini et entropy
self.min_samples_split = 2 # interessant de faire varier des fois moins séparer permet de mieux classifier
```

FIGURE 3.21 – Choix hyper-paramètres méthode de forêts aléatoires

3.6.3 Résultats

Pour visualiser le résultat on divise notre base en 80% et de validation 20% de test. Avec ce modèle nous obtenons 87% de bons résultats avec nos paramètres par défaut :

```
precision =  0.5782828282828283
```

FIGURE 3.22 – configurations initiales avec PCA

En conclusion sur cette méthode la PCA n'apporte pas des avantages.

Après cross-validation les résultats sont améliorés :

```
precision =  0.8952020202020202
```

FIGURE 3.23 – Résultats après cross validation

Nous avons aussi essayé cette méthode sur nos images, avec une précision de 45%, ici il est vraiment important de mettre un fort critère de PCA pour obtenir des meilleurs résultats.

3.7 Classification par votes de modèles

3.7.1 La méthode

Cette méthode [9] permet une prédiction à travers plusieurs classificateurs pour la prédiction .Dans la votes de modèles le choix sera basé sur le classificateur qui a eu les bons résultats entre (La méthode des K plus proches voisins, les forêts aléatoires, les

machines à vecteur de support, les réseaux de neurones) c'est-à-dire selon le poids. Le choix est basé sur le f1-score, la précision et le rappel. On a atteint un pourcentage de 75% de réussite.

3.7.2 Les hyper paramètres

Les hyper-paramètres sont les hyper-paramètres des toutes les différents modèles qu'on a déjà utilisé dans les méthodes précédentes.

```
#pour stoquer nos modèles, nous leur donnons les meilleurs hyper paramêtre de nos cross validation
self.modele_foret_aleatoire = Forets_Aleatoires(self.x_train, self.y_train, False, self.x_test, True)
self.modele_foret_aleatoire.nmb_arbre=600
self.modele_foret_aleatoire.criterion="entropy"
self.modele_foret_aleatoire.min_samples_split=4
self.pond_foret_aleatoire = 0.89
self.modele_machine_vecteurs_support = Machine_Vecteurs_Support(self.x_train, self.y_train, False, self.x_test, True)
self.pond_machine_vecteurs_support = 0.69
self.modele_machine_vecteurs_support.C = 1e-05
self.modele_machine_vecteurs_support.gamma = 'auto'
self.modele_reseaux_neuronnes = Reseaux_Neuronnes(self.x_train, self.y_train, False, self.x_test, True)
self.modele_reseaux_neuronnes.hidden_layer_sizes=(200,)
self.modele_reseaux_neuronnes.max_iter=800
self.modele_reseaux_neuronnes.random_state=None
self.pond_reseaux_neuronnes = 0.81
self.modele_knn = KNNClassifier(self.x_train, self.y_train, False, self.x_test, True)
self.modele_knn.algo="auto"
self.modele_knn.leaf_size=1
self.modele_knn.metric="manhattan"
self.modele_knn.num_neighbors=1
self.modele_knn.weights="uniform"
self.pond_knn = 0.86
```

FIGURE 3.24 – Choix hyper-paramètres méthode de votes de modèles

3.7.3 Résultats

On peut remarquer que "sans cross validation" on obtient une précision de 94,4% ce qui est très intéressant.

```
precision =  0.9444444444444444
```

FIGURE 3.25 – Résultat pour la classification par votes de modèles

4 Résultats

Par combinaison de ces modèles, on a pu avoir des résultats de classes avec meilleurs précision. Il faut indiquer le l'algorithme KNN nous a donné le meilleur résultat c'est à dire un score de 0,34 .En fait, J'ai obtenu un score de 0.80 ainsi qu'un pourcentage de test entre 94% et 95% de classification. Aussi que j'ai eu une très bonne précision mais avec un système de vote qui augmente la perte. Mais pour diminuer cette augmentation, j'ai fait une modification au niveau des hyper-paramètres pour arriver jusqu'à 0,40% de perte.

Finalement c'est avec KNN que nous avons eu le meilleur résultat sur Kaggle avec un score de 0.34 (nous cherchons à minimiser ce score).

5 Conclusion

En conclusion, A travers les classifications déjà utilisées, J'ai réussi à atteindre le maximum de précision de notre base leaf-classification à l'aide de la méthode cross validation.

La méthode de votes de modèles nous a permis de trouver meilleurs résultats en faisant combiner ces méthodes d'une façon à avoir des meilleurs résultats. Enfin notre objectif est atteint avec une précision égale à 94%.

Vous trouverez ci-joint le lien de dépôt github du code : https://github.com/Chaker-Hassen/Classification-de-feuilles-d-arbres?fbclid=IwAR2_zwSwT9LvMH1C1kOwyN5kjVowe-XfniuLGLvZap4O7L3Ymmn3FWTdvBA

Bibliographie

[1] J. Cope, T.t Beghin, P. Remagnino, & S. Barman of the Royal Botanic Gardens, Kew, UK, "Ensemble de données leaf-classification." https://www.kaggle.com/competitions/leaf-classification/data .

[2] Iden W., "Z-score et utilisation." https://medium.com/clarusway/z-score-and-how-its-used-to-determine-an-outlier-642110f3b482 .

[3] Scikit learn, "Perceptron." https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html.

[4] Scikit learn, "K plus proches voisins." https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.

[5] Scikit learn, "Combinaison de modèles, AdaBoostClassifier." https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html?highlight=adaboost%20classifier#sklearn.ensemble.AdaBoostClassifier.

[6] Scikit learn, "Support Vector Classification." https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[7] Scikit learn, " Perceptron multicouches." https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

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

[9] Scikit learn, " Voting Classifier." https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html.

[10] Autre articles format papier, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, J. Vanderplas, A. Passos and E. Duchesnay, "Scikit-learn : Machine learning in Python," *Journalof Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.