TP2 - Arbres

Alexandre CAPEL

2023-09-28

Le sujet de ce TP porte sur l'étude des arbres de décisions (ou decision tree). Nous allons apprendre à générer nos premiers arbres à partir de données simulées et enregistrées (du package `scikit-learn`) et à faire de la sélection de modèle.

Génération aritficielle de données

Durant ce TP, nous utiliserons des fonctions pour simuler ou afficher des graphiques. Ces derniers ont été empruntés au fichier `tp_arbres_source.py`.

1 Classification avec les arbres

1.1 Aparté dans le monde de la régression

Toutes les informations concernant la formalisation du problème et sa méthode de résolution (avec l'algorithme CART notamment), sont présenté dans l'énoncé du TP. Cependant, ce dernier nous donne une solution dans le cas de la classification.

En effet, on pourrait se demander quelle mesure d'homogénéité peut être utilisée dans un cadre de régression. Dans le cas de variables continues, un bon indicateur de l'hétérogénéité d'une groupe d'individu serait de calculer leur variance : en effet, cette dernière mesure le fameux "écart à la moyenne" et sera d'autant plus petite que les individus ont des valeurs proches pour cette variables.

A partir de maintenant, on se place dans le paradigme de la classification.

1.2 Premières simulations

Avec `scikit-learn`, on peut construire des arbres de décisions grâce au package `tree`. On obtient le classifieur souhaité avec la classe `tree.DecisionTreeClassifier`.

```
from sklearn import tree
```

Faisons nos premières simulations : nous allons utiliser la fonction `rand_checkers` pour construire un échantillon (équilibré) de taille $n = 456$. On peut construire nos arbres selon les deux critères présentés (entropie et indice de Gini) à l'aide du code suivant :

```
# Construction des classifieur
dt_entropy = tree.DecisionTreeClassifier(criterion='entropy')
dt_gini = tree.DecisionTreeClassifier(criterion='gini')

# Simulation de l'échantillon
n = 456
data = rand_checkers(n1=n//4, n2=n//4, n3=n//4, n4=n//4)
n_samples = len(data)
X = data[:,:2]
Y = np.asarray(data[:,-1], dtype=int)

# Entraînement des deux modèles
dt_gini.fit(X, Y)
dt_entropy.fit(X, Y)

print("Gini criterion")
print(dt_gini.score(X, Y))

print("Entropy criterion")
print(dt_entropy.score(X, Y))
```

```
Gini criterion
1.0
Entropy criterion
1.0
```

Cette classe a donc pleins d'attributs qui nous permettent d'avoir des information sur notre classifieur : ses paramètres avec `get_params()` ou sa fiabilité avec la fonction `score`.

Amusons nous à changer la profondeur maximale de l'arbre (paramètre `max_depth`), et traçon la courbe d'erreur en fonction de cette dernière. On obtient alors le graphique suivant :

On constate ici que pour le pourcentage d'erreur est quasiment nul pour une profondeur maximale égale 10. On obtiendra donc une partition très fine pour des profondeur supérieure. Cependant, il faut faire attention : nous sommes en train de regarder un pourcentage d'erreur

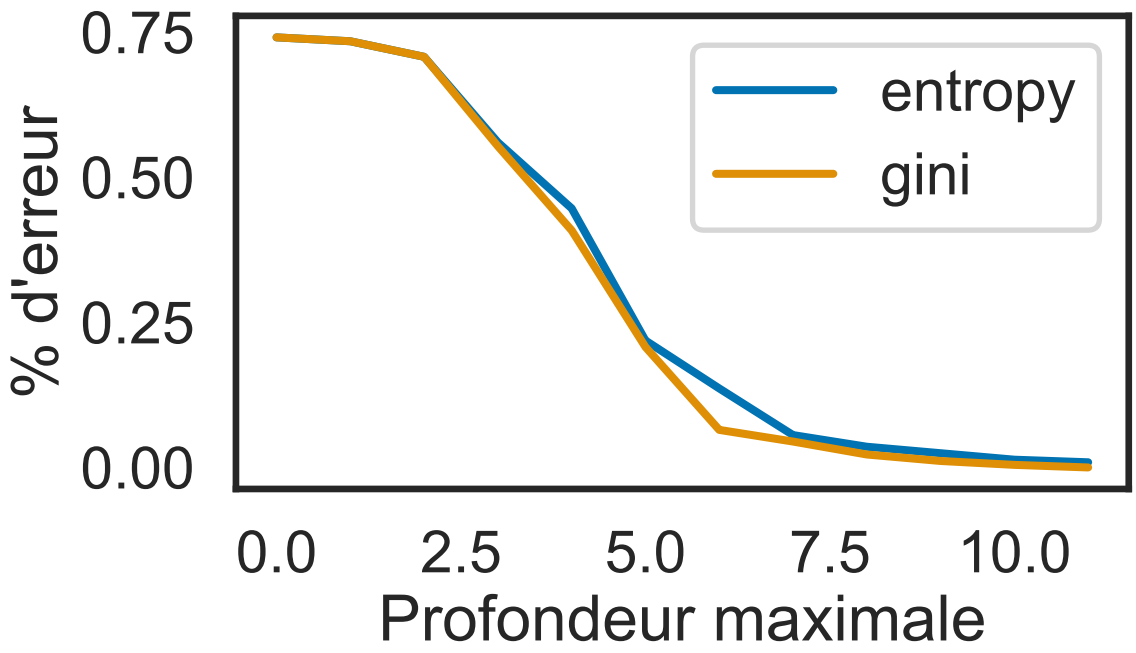


Figure 1: Pourcentage d'erreurs commises en fonction de la profondeur maximale de l'arbre

calculé à partir de nos données d'apprentissage ! Cela étant dit, il n'est pas licite d'utliser de telles valeurs pour étudier la fiabilité de notre classifieur.

En effet, étudions ce qu'il se passe dans le cas où l'on construit une parition de taille égale à la taille de notre échantillon. On obtiendra alors une parition dont chaque sous ensemble contient un seul individu. Impossible pour l'arbre de décision de se tromper… en ce qui concerne les données d'apprentissages ! Néanmoins, un tel classifieur ne sera pas adapté à la prédiction puisqu'il sera trop lié aux données avec lequel il a appris : on est dans un cadre typique de sur-apprentissage.

L'objectif va donc être de trouver une profondeur idéale pour ne pas sur-apprendre, tout en étant assez grande pour ne pas avoir un biais trop important (cas de sous-apprentissage).

Mais, nous pouvons quand même utiliser cette courbe pour en tirer des conclusions. On peut voir tout d'abord que pour les deux critères, les courbes sont similaires. D'autre part, ces-dernières tombent assez rapidement à une valeur proche de 0, ce qui nous conforte dans la pertinence de ce classifieur (cela siginifie que notre modèle apprend).

Regardons comment notre arbre paritionne l'espace avec la fonction `frontiere`.

```
dt_entropy.max_depth = np.argmin(1-scores_entropy)+1
plt.figure(figsize=(6,3.2))
frontiere(lambda x: dt_entropy.predict(x.reshape((1, -1))), X, Y, step=100)
plt.draw()
print("Best scores with entropy criterion: ", dt_entropy.score(X, Y))
```

```
Best scores with entropy criterion:  0.9910714285714286
```

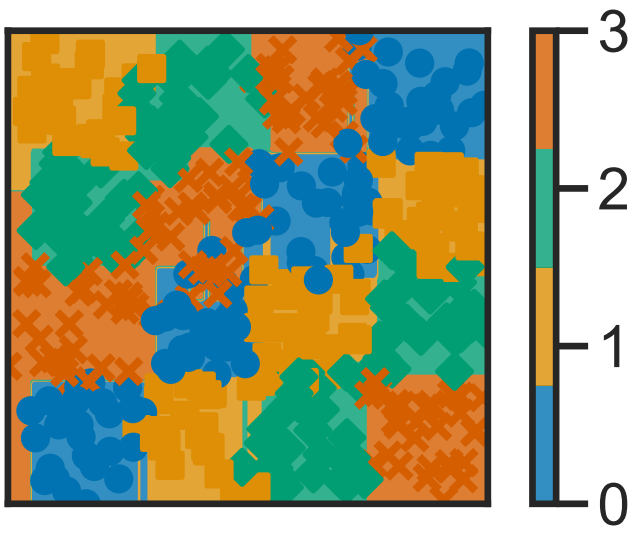


Figure 2: Frontières pour la meilleur profondeur (entropie)

La partition est en effet très bonne, mais on constate une variance importante expliquant l'erreur proche de 0. Mais le plus intéressant dans ce graphique est la manière dont sont disposées nos données simulées, et pour l'expliquer, nous allons devoir nous pencher sur comment sont construites ces données.

Déviation sur la `rand_checkers`

En étudiant plus précisément la fonction `rand_checker`, on voit que nos données sont construites de manière quadrillée, dans l'espace, à une erreur gaussienne près. La structure même de ces données explique le bon apprentissage de notre classifieur : les données semblent "collées" parfaitement à la méthode de paritionnement utilisé par l'algorithme CART.

Nous avons donc produit un bon paritionnement de l'espace, seulement il est fréquent que les données soient plus complexes et vivent en dimension plus grande. Un moyen pour pouvoir visualiser l'arbre de décisions est de télécharger ce dernier avec `graphviz`. Pour le modèle sélectionné on obtient :

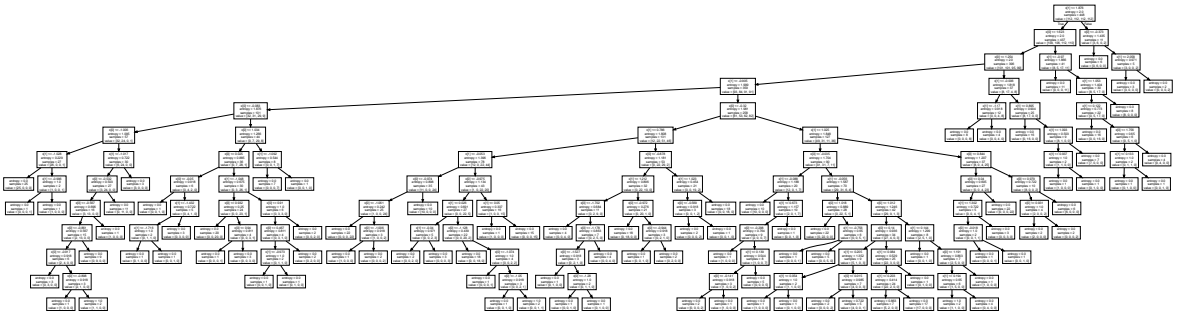


Figure 3: Arbre obtenu

Regardons un modèle plus lisible (avec moins de profondeur) pour comprendre la structure du graphique:

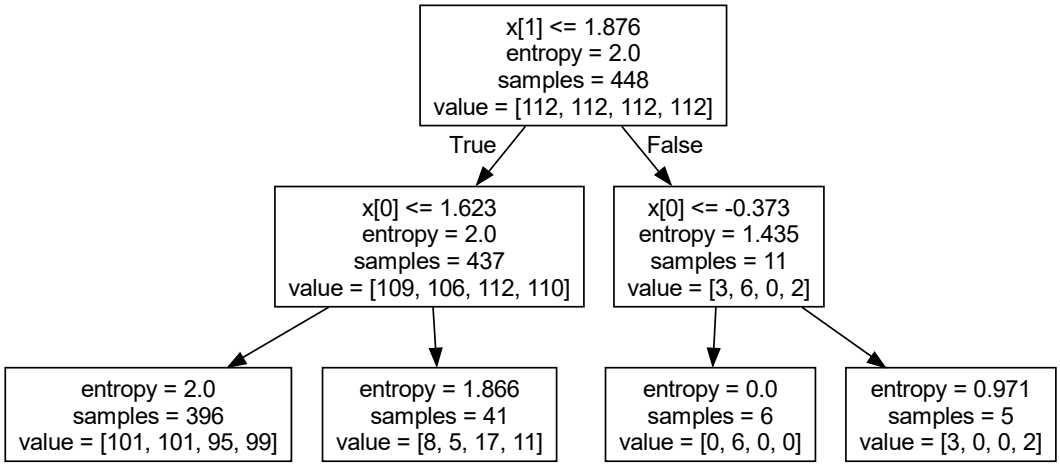


Figure 4: Arbre simplifié de profondeur 2

Dans Figure 4, on remarque qu'il y a des informations qui sont écrites pour chaque noeud (correspondant à un sous-ensemble plus ou moins grand) :

- la première ligne correspond au seuil donné à une des variables pour prendre une décision (gauche si la condition est vérifiée, droite sinon). C'est pourquoi cette ligne n'esixte pas sur les noeuds terminaux.
- la valeur du critère (ici entropie) pour les variables présente dans le sous-ensemble
- le nombre d'individu total dans le sous-ensemble
- la répartition des individus pour chaque groupe

Par exemple, si un individu a pour variable $x = (2, -1)'$, on décidera dans le premier noeud d'aller au noeud intermédiaire de droite dans un premier temps, puis finir par celui de gauche. Le groupe décidé sera celui qui prédomine dans le sous ensemble : dans notre exemple, on voit que ce sera le groupe 2.

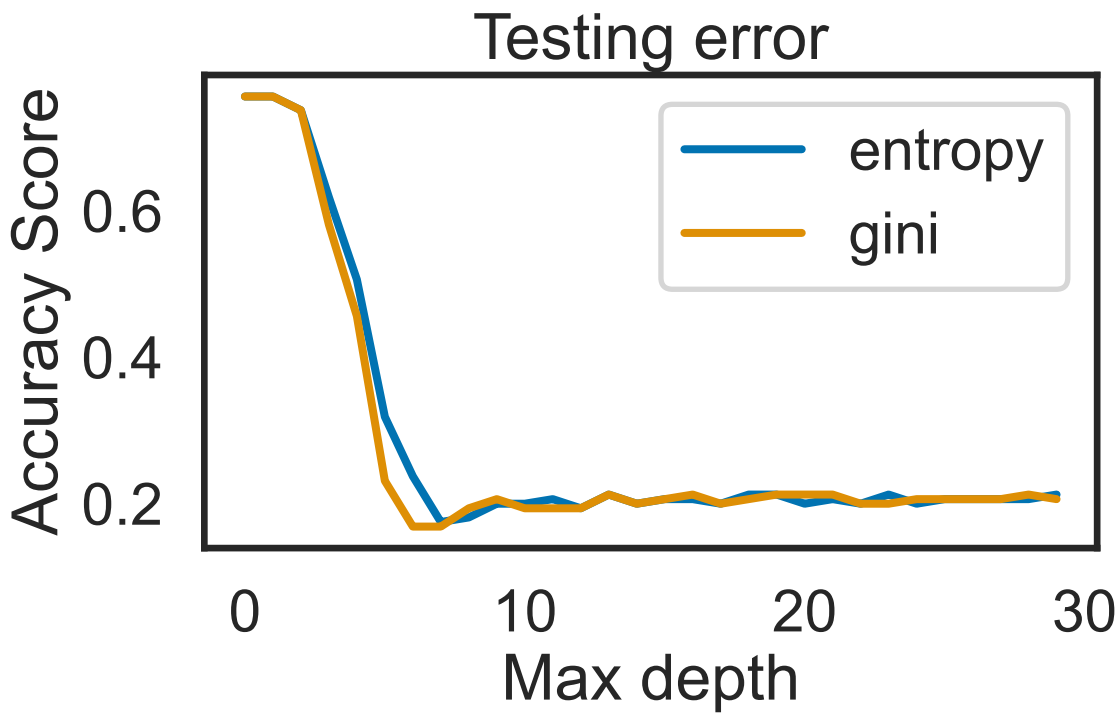
1.3 Etude de l'erreur sur un échantillon test

Maintenant, essayons de regarder la véritable fiabilité de notre arbre en regardant la proportion d'erreur faite par ce dernier sur de nouvelles données simulées pour $n = 160 = 40+40+40+40$, avec `rand_checker`.

```
# Initialisation de l'échantillon de test
data_test = rand_checkers(n1=40, n2=40, n3=40, n4=40)
X_test = data_test[:,:2]
Y_test = np.asarray(data_test[:,-1], dtype=int)
```

Nous allons tracer une nouvelle courbe d'erreur en fonction de la profondeur de l'arbre :

```
Text(0.5, 1.0, 'Testing error')
```



Ici, on constate une nette chute au début pour les petites valeurs de la profondeur puis une stabilisation au bout d'un certain nombre : on voit ici qu'il est inutile d'augmenter la profondeur de l'arbre, le taux d'erreur ne semble ne pas vouloir s'améliorer.

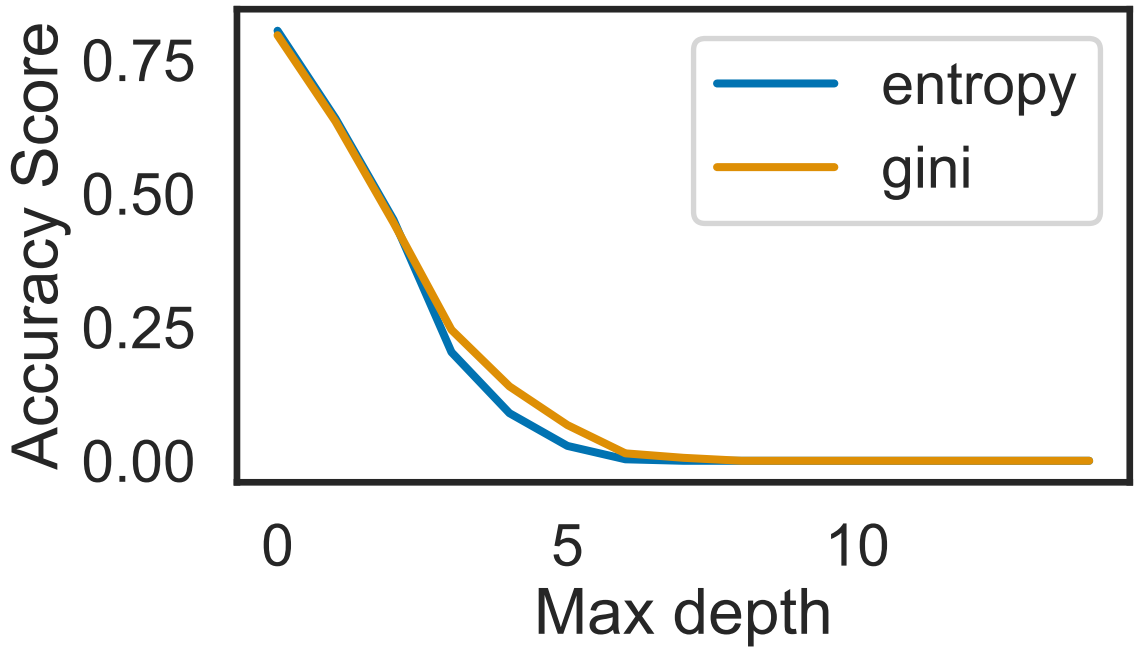
1.4 Nouvelle analyse sur les données `DIGITS`

Nous allons retracer les graphiques précédents sur des données disponible dans le module `sklearn.datasets`: le jeu de données `DIGITS`.

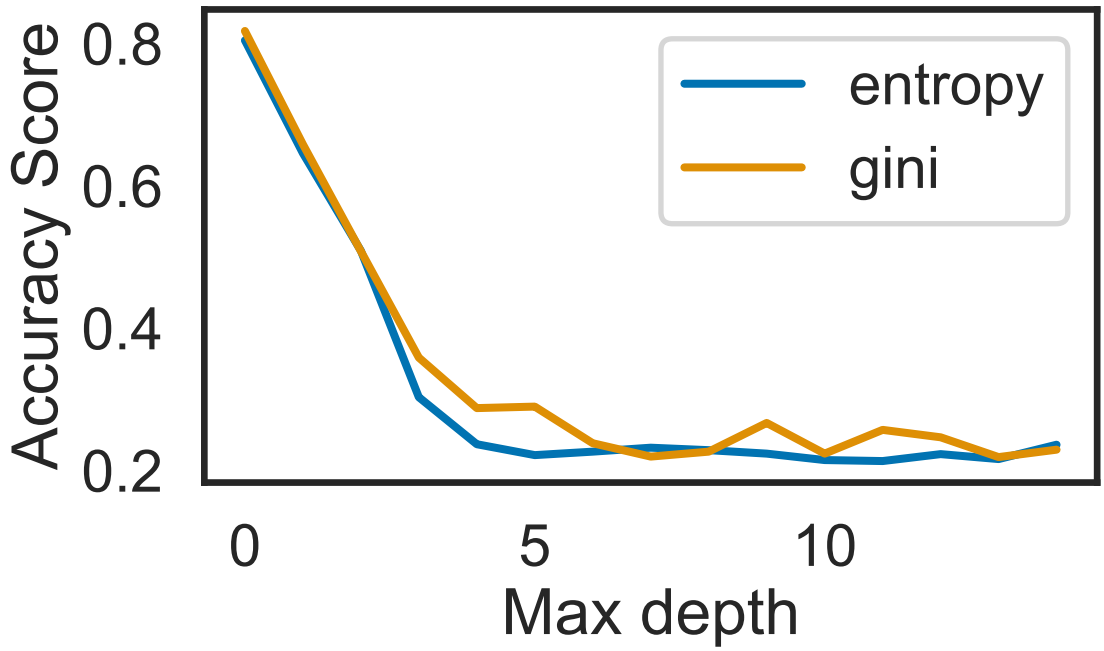
A la différence du cas simulé, nous ne pouvons pas produire nous même des données de test. Un moyen de contourner le problème est de paritionner l'ensemble du jeu de données et un échantillon d'apprentissage servant à entraîner le modèle, et un échantillon de test servant à regarder la fiabilité du modèle. On a fait le choix de faire une découpe test/train de taille 80% - 20%.

```
digits = datasets.load_digits()          # Chargement des données
n_samples = len(digits.data)
X_train, X_test, Y_train, Y_test = model_selection.train_test_split(digits.data,
                                                                    digits.target,
                                                                    test_size=0.8,
                                                                    random_state=50)
```

Ici, la variable `random_state` permet de fixer une graine pour avoir toujours les mêmes découpages en test/train. Comme précédemment, on peut maintenant calculer la courbe d'erreur sur les données d'apprentissage en fonction de la profondeur maximale pour les deux critères.



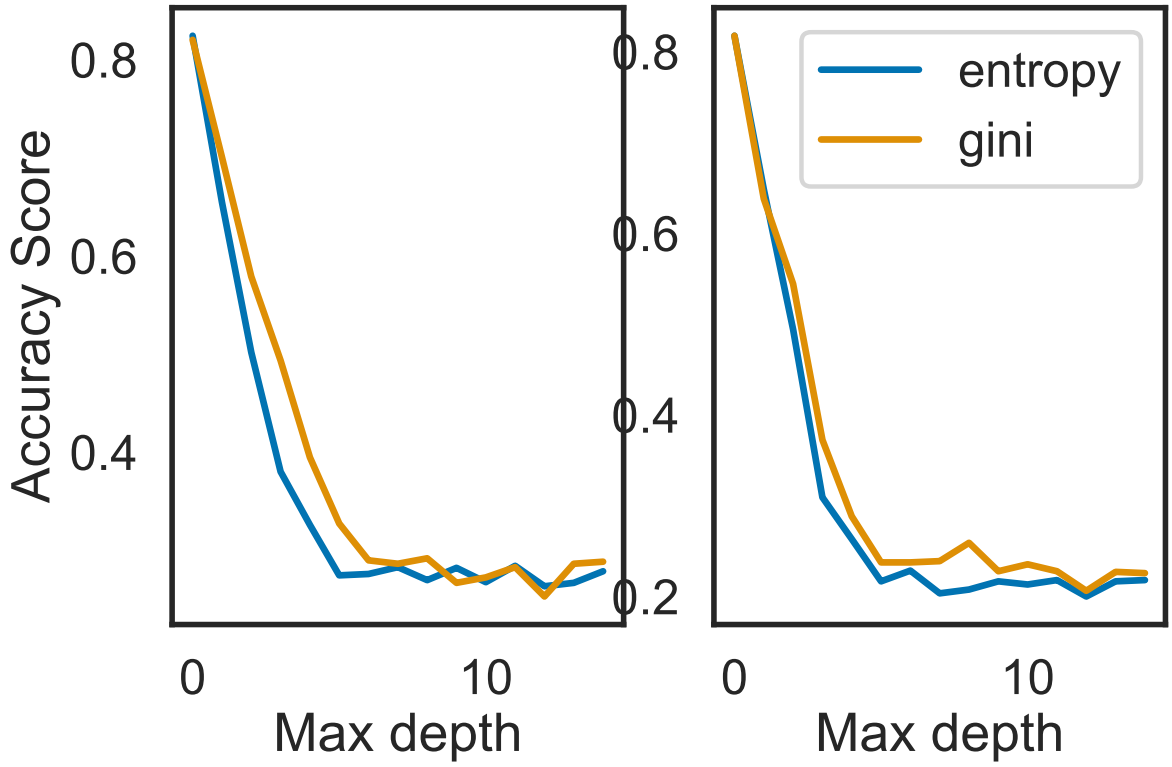
Mais ce qui nous intéresse c'est bien sur la courbe d'erreur pour les données de test :



Etudions maintenant cette courbe. Comme vu plus haut, on obtient essentiellement le même style d'allure, avec une nette chute sur les premières valeurs puis une stabilisation à partir

d'un certain rang. On pourrait avoir la tentation de dire que le meilleur modèle est celui avec la profondeur maximale égale à 6. Cependant, il ne faut pas être aussi hâtif !

Regardons deux autres courbes en changeant la parition train/test (modification du paramètre `random_state` dans `train_test_split`). On obient alors les courbes :



Ici, on remarque que :

- les courbes sont différentes en fonction de la partition
- les minimums ne sont pas atteint par la même profondeur
- cette valeur minimale de l'erreur est différente

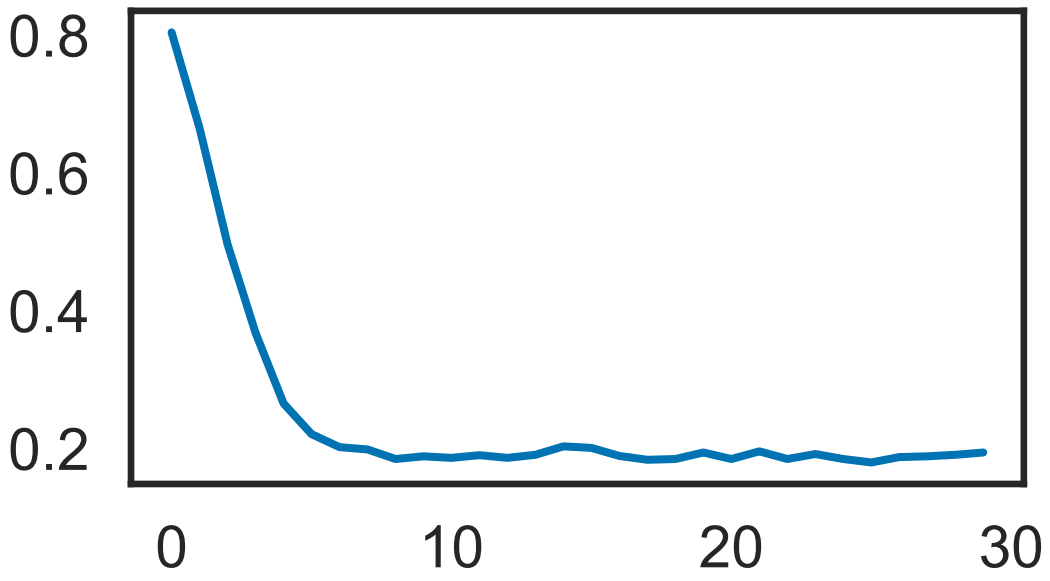
Pour pouvoir sélectionner un modèle selon ce critère, nous allons devoir faire en sorte de nous affranchir de cette sensibilité de partitionnement des données. Un moyen de le contourner est de faire de la validation croisée.

2 Méthode de choix de paramètres - Sélection de modèle

Nous allons donc faire de la validation croisée 5-fold. Pour ce TP, nous allons utiliser la fonction `cross_val_score` du module `sklearn.model_validation`.

```
# Initialisation avec graine fixée
np.random.seed(123)
dmax = 30
X = digits.data
Y = digits.target
error = np.zeros(dmax)

# Boucle de calcul de l'erreur
for i in range(dmax):
    dt_entropy = tree.DecisionTreeClassifier(criterion='entropy', max_depth=i+1)
    error[i] = np.mean(1-cross_val_score(dt_entropy, X, Y, cv=5))

# Affichage de la courbe
plt.figure(figsize=(6,3.2))
plt.plot(error)
best_depth = np.argmin(error)+1
print("Best depth: ", best_depth)
```

```
Best depth:  26
```



Comparée aux courbes d'erreur précédentes, celle-ci semble plus régulière (sûrement un effet de la moyenne). On constate également que l'arbre minimisant l'erreur moyenne est celui de profondeur maximale égale à 26, ce qui est bien différent de ce que l'on avait dans la section

précédente. Néanmoins, on remarque que l'on retrouve des taux d'erreur d'ordre similaire aux alentours de 10 : il serait peut être préférable de prendre un arbre de profondeur maximale vers cette valeur (pour l'interprétation par exemple). On choisira donc cette valeur égale à 8 ici.

Voici l'arbre obtenu :

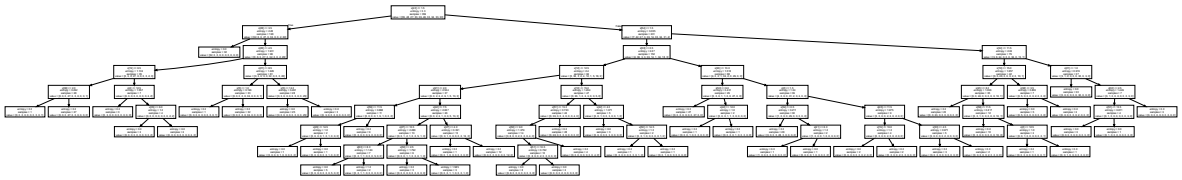


Figure 5: Meilleur arbre

Courbe d'apprentissage

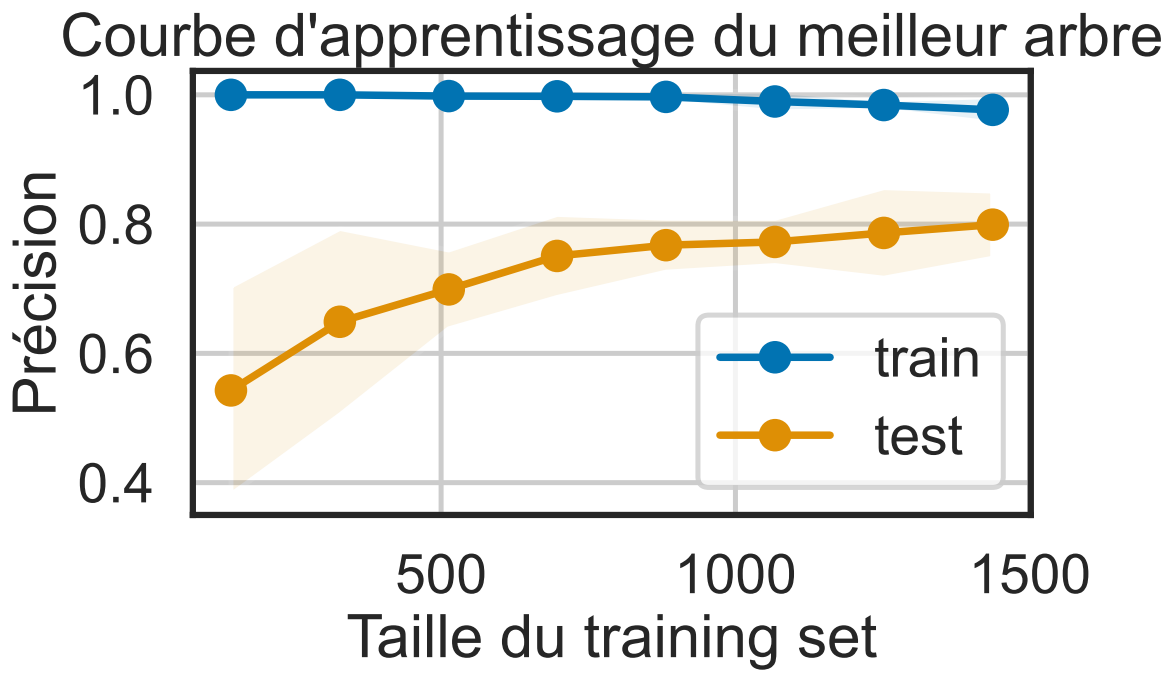
Nous allons terminer ce TP en traçant la courbe d'apprentissage de notre arbre de décision optimal de profondeur maximale égal à 8.

Cette courbe nous permet de voir rapidement si le modèle sélectionné est raisonnable, dans le sens où on ne se situe pas dans un cadre de sur ou sous-apprentissage. Pour obtenir cette courbe nous allons utiliser la fonction `learning_curve` du module `sklearn.model_selection`.

```
# Calcul des courbes d'apprentissage
size = np.linspace(0.1, 1, 8)
dt = tree.DecisionTreeClassifier(criterion='entropy', max_depth=8)
n_samples, train_curve, test_curve = learning_curve(dt, X, Y, train_sizes=size)

# Mise en place de l'affichage
plt.figure(figsize=(6,3.2))
plt.grid()

# colorisation des intervalles de confiance
plt.fill_between(n_samples,
                 np.mean(train_curve, axis=1) -1.96*np.std(train_curve, axis=1),
                 np.mean(train_curve, axis=1) + 1.96*np.std(train_curve, axis=1),
                 alpha=0.1)
plt.fill_between(n_samples,
                 np.mean(test_curve, axis=1) -1.96*np.std(test_curve, axis=1),
                 np.mean(test_curve, axis=1) + 1.96*np.std(test_curve, axis=1),
                 alpha=0.1)

# Affichage des courbes
```

```
plt.plot(n_samples, np.mean(train_curve, axis=1),"o-", label="train")
plt.plot(n_samples, np.mean(test_curve, axis=1), "o-", label="test")
plt.legend(loc="lower right")
plt.xlabel("Taille du training set")
plt.ylabel("Précision")
plt.title("Courbe d'apprentissage du meilleur arbre")
```

```
Text(0.5, 1.0, "Courbe d'apprentissage du meilleur arbre")
```



Dans figure, nous avons deux courbes :

- la première (en bleue), représentant le score des données d'apprentissage, semble assez constante avec une valeur proche de 1 : on voit typiquement que notre modèle a très bien appris sur nos donénes d'apprentissage et que donc le cas de sous-apprentissage est peu probable.
- la seconde (en orange), représentant le score de validation croisée, est croissante et semble atteindre une valeur proche des meilleurs scores : on voit ici que notre modèle se généralise très bien, ce qui laisse penser que nous ne sommes pas non plus dans un contexte de sur-apprentissage.

3 Conclusion

Dans ce TP, nous avons pu comprendre en profondeur le fonctionnement des arbres de décisions et de la manière dont ces derniers sont implémentés sur `sklearn`. De plus, on a pu utiliser de nouveaux outils pour valider nos modèles, comme les fontions de validation croisée et de courbe d'apprentissage, pouvant être utilisées pour d'autres modèles autres que les arbres de décision.