Université de Strasbourg

Intelligence Artificielle

Licence 3 Informatique

TP : Classification d'Iris avec un arbre de décision

L'objectif de ce TP est de mettre en œuvre, *from scratch*, un modèle "arbre de décision" pour la classification d'Iris (oui, les fleurs [a])

Objectifs spécifiques

- Observer les données et comprendre leur nature pour les adapter (si besoin) à leur utilisation dans un arbre de décision.
- Comprendre le fonctionnement d'un arbre de décision afin d'en proposer une mise en œuvre dans un programme.
- Observer les résultats (à défaut d'évaluer le modèle, ce qui sera l'objet d'un cours futur) et les corréler avec ce qu'on sait des données.

a. `https://en.wikipedia.org/wiki/Iris_(plant)`

1 Les données

Durant ce TP, nous allons travailler sur un problème de classification (*i.e.* assigner des étiquettes à des données de façon à les grouper dans des catégories distinctes). Le jeu de données que nous allons utiliser est traditionnellement utilisé pour se faire la main en apprentissage automatique : c'est le jeu de données des Iris. Il s'agit d'un jeu de données multi-varié permettant de décrire 3 espèces d'Iris. En tout, le jeu comprends 50 instances de chaque espèce (il y a donc un total de 150 instances –vraiment peu). Chaque instance est décrite par 4 attributs (*features*, descripteurs, traits ou caractéristiques) réels et est associée à une étiquette (la classe à laquelle l'instance appartient). Ces 4 attributs sont les suivants : *sepal_length*, *sepal_width*, *petal_length* et *petal_width*.
La figure 1 illustre une manière (parmi d'autres) de visualiser ces données. Le fait est que ces données sont de faible dimension ("seulement" 4) et que nous avons seulement 150 instances. Dans ce cas, ça n'est pas trop difficile de proposer une visualisation des données (à l'inverse de pas mal de données du "monde réel"). On peut bien entendu utiliser d'autres techniques de visualisation, comme des diagrames, des histogrammes, etc.

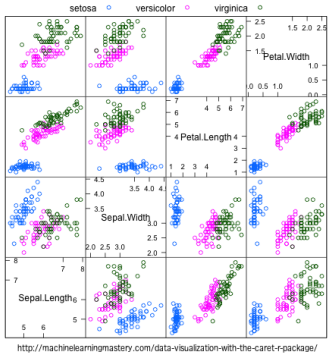

Figure 1 – Visualisation des données dans une *scatterplot matrix* (un nuage de point)

Nous pouvons observer que les données peuvent en effet être distinguées par leur classe (les espèces). De manière intéressante, on note que 2 groupes principaux émergent : "Setosa" *v.s.* "Versicolor" et "Virginica" : la classe "Setosa" est dite linéairement séparable des deux autres classes (autrement dit, une simple droite permet de classer des entrées en "Setosa" et "non-Setosa"). Distinguer entre "Versicolor" et "Virginica" est moins trivial (du moins graphiquement) : si nous utilisions une méthode de *clustering* non supervisé, les groupes obtenus pourraient être différents de ce à quoi on s'attend (on en dira plus lors du TP sur K-means).

★ *Question de synthèse (on s'échauffe gentiment) : est-ce qu'un arbre de décision est utilisé dans un contexte d'apprentissage supervisé ou non supervisé ? Expliquez votre réponse.*

2 Import des données et visualisation (basique)

On propose d'utiliser la bibliothèque python Pandas (`https://pandas.pydata.org/`) pour charger les données (*NB* ça n'est pas la seule manière de faire, loin s'en faut, mais c'est une bibliothèque assez répandue donc ça vaut la peine de l'utiliser un peu)

Pour charger des données depuis un csv, il faut utiliser la méthode `read_csv()`. Les données sont importées dans un *dataframe*[1] (terminologie de Pandas) et un certain nombre de méthodes sont ensuite utilisable sur ce dataframe, comme la méthode `head()` qui permet d'afficher les 5 premières instances du jeu de données ou encore la méthode `info()` permettant d'afficher des informations sur les attributs du jeu de données.

★ *Utilisez le snippet de code suivant pour charger les données (le fichier csv est à télécharger sur Moodle) et afficher les 5 premières instances*

```
import pandas as pd

root = "./"

iris_dataframe = pd.read_csv(root + 'iris.csv')
print(iris_dataframe.head())
print(iris_dataframe.info())
```

Pour visualiser, il existe une multitude de solutions également (Matplotlib ou encore Seaborn[2], qui repose sur matplotlib). L'objet de ce cours n'est (malheureusement ?) pas de travailler sur des questions de visualisation, mais je vous propose de reproduire le *scatterplot* de la figure 1 avec Seaborn.

★ *Utilisez le snippet suivant pour visualiser les données dans un* scatter plot

```
import seaborn as sns

sns.pairplot(iris_dataframe, hue='class')
```

3 Construction d'un arbre de décision

On rappelle que chaque instance du jeu de données est décrite par 4 attributs (+ 1 étiquette, la classe de l'instance). Construire un arbre de décision, c'est déterminer lequel de ces 4 attributs a le plus grand pouvoir discriminant (autrement dit, quel attribut permet le plus "facilement" de dire "cette instance est une Iris-truc, cette instance est une Iris-machin"). On parle aussi de *gain d'information*, *i.e.* quel attribut apporte le plus d'information pour décrire les données et les comprendre.

Une fois déterminé l'attribut le plus discriminant, on poursuit le processus avec les 3 attributs restant : pour raffiner la classification, il faut déterminer lequel des attributs restant est le plus discriminant, *etc.* Pour vous arrêter, soit vous épuisez tous les attributs (pas génial parce qu'on va vraissemblablement faire du sur-apprentissage), soit on fixe un autre critère (profondeur de l'arbre, proportion de chaque classe pour un attribut donné, ... –on y reviendra tout à l'heure).

Une fois l'arbre construit, on souhaite pouvoir y examiner une nouvelle instance (pour laquelle on n'a pas d'étiquette) et prédire à quelle classe cette instance appartient.

★ Question préliminaire

1. Comment proposez-vous de gérer les valeurs réelles décrivant les données de façon à construire un arbre de décision ?

1. `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html`
2. `https://seaborn.pydata.org/index.html`

On propose de partir sur la construction d'un arbre de décision binaire : à chaque noeud, on prendra une décision "oui" / "non" selon la branche empruntée (est-ce qu'il fait soleil ou non ; est-ce qu'il pleut ou non ; est-ce que l'hygrométrie est inférieure à 60% ou non, etc.). Dans le cas de notre jeu de donnée, les questions que l'on va poser à chaque noeud de l'arbre seront du type "est-ce que la largeur du pétal est inférieur à x cm ?" et les données vont être partitionnées en deux selon la réponse à cette question. Il faudra donc être capable de **déterminer la valeur** x (dénotée dans la suite par "valeur de *split*") afin de partitionner les données.

Notez que le nombre de branches partant de chaque nœud n'a pas nécessairement à être deux : rien ne dit qu'un arbre de décision doit être binaire.

3.1 Mesure de l'entropie d'une partition

Mettez en œuvre la fonction `entropie(dataframe)` qui calcule et retourne l'entropie des données passées en paramètres –l'ensemble du jeu de données ou bien une des partitions ... le fonctionnement est strictement similaire) (*NB* : les formules pour les calculs d'entropie et de pouvoir discriminants sont rappelées en annexe).

Quelques tips Python pour vous aider :

```
# Nombre d'instances dans le dataframe:
nb_lignes = dataframe.shape[0]

# Decompte du nombre d'instance de chaque classe dans le dataframe
series = dataframe['class'].value_counts()
# Sur le dataframe global (l'ensemble des donnees), series retourne:
# Iris-setosa        50
# Iris-versicolor    50
# Iris-virginica     50
# Name: class, dtype: int64

# Recuperer une valeur associee a une etiquette donnee
occ_setosa = series.get('Iris-setosa')
```

3.2 Mesure du gain d'information des attributs

Ici, chaque évaluation du gain d'information devra se faire sur le jeu de donnée trié sur l'attribut examiné (et le processus sera répété pour chaque attribut, afin de déterminer quel attribut est le plus discriminant). Pour trier le dataframe selon un attribut, vous pouvez utiliser (en l'adaptant) le code suivant :

```
# Exemple de tri du dataframe sur l'attribut 'petal_length'
data_sorted = iris_dataframe.sort_values(by='petal_length')
```

Une fois les données triées, il faut déterminer la valeur de *split*. Cela peut être fait de différentes manières :

- la plus triviale (et pas nécessairement la plus efficace) est de simplement prendre la valeure moyenne (voire médiane) de l'attribut dans la partition. Ensuite, on crée deux partitions : dans la première, on met toutes les instances dont la valeur sur l'attribut examiné est inférieure à la valeur de *split*, x ; dans la seconde, toutes les autres instances.
- on peut raffiner ce split en utilisant des quantiles : là aussi, on va construire deux partitions en mettant dans la première les instances dont la valeur sur l'attribut examiné est inférieure à la valeur de *split* correspondant au *n-ième* quantile et dans la seconde toutes les autres instances. Ensuite, on mesure le gain d'information pour chacun des splits et on retient la valeur de split (et les partitions associées) pour laquelle on a obtenu le gain le plus important.

— on peut aller encore plus loin et tester chacune des valeurs possible de l'attribut pour faire le split (en travaillant sur des valeurs réelles, inutile de dire qu'il peut y avoir beaucoup de split à tester !)
— enfin, la dernière méthode proposée ici est de parcourir les instances une à une et de détecter un changement de classe : à la détection d'un tel changement, on prend la valeur de l'attribut examiné comme valeur de split et on crée les deux partitions ($< x$ et $\geq x$). Pour mieux comprendre cette idée, ouvrez le csv dans un logiciel de tableur (libreoffice calc par exemple), triez les données sur l'attribut `petal_length` : vous observerez qu'en dessous de `petal_length` = 3cm, toutes les instances sont des Setosa. Le split est parfait ici : toutes les instances de Setosa seront dans la première partition et les autres instances (qu'il faudra donc continuer à examiner pour les classer) seront dans la seconde partition.

Vous pouvez utiliser la méthode de votre choix pour la suite du TP. Il est cependant conseillé de suivre la dernière méthode pour pouvoir comparer votre arbre à l'arbre ci-dessous, construit en utilisant le split sur le changement de classe.

L'algorithme pour calculer le gain des partitions est le suivant (utilise le critère de changement de classe pour construire les 2 partitions) :

Algorithme 1 : Calcul du gain pour un attribut donné

Données :
data : dataframe contenant les données à partitionner
a : attribut sur lequel vous travaillez

```
1  E ← entropie(data)
2  gain ← 0
3  split_value ← 0
4  partitions ← []
5  sorted_data ← données triées selon l'attribut a
6  classe ← sorted_data['class'][0]
7  pour chaque instance i de sorted_data faire
8      si classe de l'instance i ≠ classe alors
9          split_value ← valeur de l'attribut a de l'instance i
10         partitions[0] ← données pour lesquelles a < split_value
11         partitions[1] ← données pour lesquelles a ≥ split_value
12         gain ← calcul du gain pour l'attribut a
13         break
14     fin
15 fin
16 retourner (gain, split_values, partitions)// tuple python
```

Il faut encore déterminer quel attribut est le meilleur mais le plus dur est déjà fait : il ne reste qu'à itérer sur les différents attributs du jeu de données et de retourner le meilleur attribut, celui qui a le gain le plus important. Vous pouvez soit appeler la fonction ci-dessus dans une autre fonction, soit rajouter une boucle dans la fonction ci-dessus.

Dans un cas comme dans l'autre, votre fonction doit renvoyer : le meilleur attribut, son gain, sa valeur de split et les partitions associées.

3.3 Construction de l'arbre de décision

En premier lieu, vous allez avoir besoin de définir une classe pour représenter un nœud de votre arbre. Chaque nœud est caractérisé par un attribut, une valeur de split et, si c'est une feuille, d'une prédiction (la proportion de chaque classe dans le nœud –si le nœud est pur, il n'y aura qu'une classe. S'il ne l'est

pas, il y en aura plusieurs et la proportion d'instances de chaque classe permettra de déterminer quelle classe est la plus probable). Chaque nœud aura aussi une branche droite et une branche gauche ; elles seront construites récursivement.

★ *Pourquoi tous les nœuds de l'arbre de décision ne seront pas nécessairement purs ?*

Pour rappel, l'expression `data['class'].value_counts()` retourne le nombre d'instances pour chaque classe possible de `data`. Pour un nœud donné, c'est cette méthode qui permettra d'enregistrer la "prédiction".

La construction de l'arbre va se faire récursivement. Il faut donc déterminer le critère d'arrêt de l'algorithme. La première possibilité, la plus évidente, est de s'arrêter lorsque l'on a épuisé tous les attributs. Une autre possibilité est de s'arrêter lorsque la profondeur de l'arbre dépasse un certain seuil (cela présente en plus l'avantage de limiter le sur-apprentissage du modèle –les arbres de décision sont des modèles très sujets au sur-apprentissage).

Lorsque le critère d'arrêt est atteint, cela signifie qu'une feuille de l'arbre doit être créée. Sinon, il faut continuer à construire les branches (gauche/droit) récursivement. L'algorithme de construction de l'arbre est le suivant :

Algorithme 2 : Construction de l'arbre de décision (fonction `construction_arbre`, récursive)

```
attribut, gain, split, partitions ← meilleur_attribut()          // fonction de la partie 2.2
prediction ← data[cible]value_counts
si profondeur > seuil ou attributs_restants est vide alors
    retourner Noeud(prediction=prediction, feuille=True)
fin
/* attributs_restants est la liste des attributs non encore retenus : il faut
   donc passer en argument de l'appel récursif la liste des attributs de
   laquelle vous avez ôté l'attribut retourné ci-dessus                    */
branche_gauche ←
 construction_arbre(partitions[0], cible, attributs_restants, profondeur+1)
branche_droite ←
 construction_arbre(partitions[1], cible, attributs_restants, profondeur+1)
retourner Noeud(split, attribut, branche_gauche, branche_droite, prediction)
```

Cette fois vous avez fait le plus dur. Pour construire votre arbre, vous pouvez simplement appeler :

```
arbre = construction_arbre(data, 'class',
            ['sepal_length', 'sepal_width', 'petal_length', 'petal_width'], 0)
```

Voici une fonction qui vous permettra d'afficher votre arbre (possiblement à modifier un peu selon les structures de données que vous avez définies) :

```
def print_tree(node, spacing=''):
    if node is None:
        return
    if node.isLeaf:
        print(spacing + node.node_result(spacing))
        return
    print('{}[Attribute: {}  Split value: {}]'.format(spacing, node.attribute, node.
    split_value))

    print(spacing + '> True')
    print_tree(node.left_branch, spacing + '-')

    print(spacing + '> False')
    print_tree(node.right_branch, spacing + '-')
    return
```

Cet affichage devrait vous donner quelque chose comme ceci :

```
[Attribute: petal_length  Split value: 3.0]
> True
- Class 0 Count: 50
-
> False
-[Attribute: petal_width  Split value: 1.4]
-> True
-- Class 1 Count: 28
--
-> False
--[Attribute: sepal_length  Split value: 5.2]
--> True
--- Class 2 Count: 1
---
--> False
--- Class 2 Count: 49
--- Class 1 Count: 22
---
```

3.4 Inférence de la classe de nouvelles instances

Normalement, pour tester on utilise des données autres que celles utilisées pour construire nos algorithmes (ce que l'on a pas fait ici... mais ça n'est pas bien grave).

L'instruction python suivante permet de prélever, aléatoirement, une instance dans un dataframe :

```
1 instance = data.sample()
```

Pour inférer la classe de cette instance, il faut parcourir l'arbre (récursivement, encore) selon l'algorithme suivant :

Algorithme 3 : Inférence de la classe de nouvelles instances (fonction `inference`, récursive)

```
1  si noeud est une feuille alors
2  |  afficher le résultat de la prédiction
   |  /* utiliser la prédiction enregistrée dans le noeud pour être capable de dire
   |     n% de chance d'être de classe A; m% de chance d'être de classe B, etc.   */
3  fin
4  sinon
5  |  valeur_attribut ← valeur de l'attribut enregistré dans le noeud pour l'instance (e.g. valeur
   |  de l'attribut x_i de l'instance)
6  |  si valeur_attribut < valeur de split du noeud alors
7  |  |  inference(instance, noeud.branche_gauche, noeud.attribut)
8  |  fin
9  |  sinon
10 |  |  inference(instance, noeud.branche_droite, noeud.attribut)
11 |  fin
12 fin
```

4 Annexes

4.1 Entropie et pouvoir discriminant

Entropie d'un ensemble de données Soit n_k le nombre d'instances appartenant à une classe $k \in K$ dans le dataframe et |data| le nombre total d'instances dans le dataframe `data`. L'entropie de ce dataframe est :

$$H(\text{data}) = -\sum_{k=0}^{K} \left(\frac{n_k}{|\text{data}|} \right) log_2 \left(\frac{n_k}{|\text{data}|} \right) \tag{1}$$

Gain d'information d'un attribut i Soit $|p|$ le nombre d'instances dans une partition $p \in P$, $H(p)$ l'entropie d'une partition $p \in P$, $H(\text{data})$ l'entropie totale du jeu de données `data`, et |data| le nombre total d'instances dans le jeu de données. Le gain d'un attribut i est défini par :

$$\text{Gain}(i) = H(\text{data}) - \left(\sum_{p=0}^{P} \left(\frac{|p|}{|\text{data}|} \right) H(p) \right) \tag{2}$$