M2 ID APP
LANGAGES DE SCRIPT
TRAVAUX DIRIGÉS
Brice Mayag

ARBRES DE DÉCISION EN PYTHON

Les arbres de décision font parties des méthodes d'apprentissage automatique les plus simples. Il s'agit d'une méthode totalement transparente de classification d'observations, qui, après apprentissage, ressemble à une suite d'instruction "Si-Alors" présentée sous-forme d'un arbre. Une fois l'arbre créé, il est assez facile de comprendre le processus de décision. Il suffit de suivre le chemin vers les feuilles de l'arbre, en répondant correctement aux questions, pour y arriver à une réponse.

Nous allons expliquer comment représenter un arbre de décision en langage python. La première étape consiste créer une classe nommée `decisionnode`, qui représente un noeud de l'arbre :

```
class decisionnode:
 def __init__(self,col=-1,value=None,results=None,tb=None,fb=None):
  self.col=col
  self.value=value
  self.results=results
  self.tb=tb
  self.fb=fb
```

Chaque noeud possède 5 variables d'instances, dont les valeurs peuvent être fixées dans la méthode d'initialisation :

- `col` : indique l'indice de colonne du critère à tester ;
- `value` : précise la valeur de la colonne qui correspond à un résultat vrai ;
- `results` : contient un dictionnaire des résultats de la branche en cours. Sa valeur est `None`, excepté lorsque le noeud est une feuille.
- `tb` et `fb` sont des `decisionnode` (n ?uds de décision), qui correspondent aux noeuds suivants de l'arbre selon que le résultat est, respectivement, Vrai ou Faux.

Prédire le comportement d'un utilisateur pour un site web

Le site internet `www.m2idapp.com` propose une application en ligne payante, appelée *ProgPyth*, de programmation en python. Pour découvrir cette offre, les internautes peuvent s'inscrire et l'utiliser *ProgPyth* pendant une durée limitée de 24h, durée après laquelle ils peuvent opter pour un service basique ou premium.

Les administrateurs du site internet souhaitent déterminer, de façon automatique, parmi les utilisateurs inscrits, ceux qui vont effectivement devenir des abonnés payants (i.e. ceux qui auront choisi la formule basique ou premium après avoir essayé l'application). Pour cela, chaque internaute doit fournir un certain nombre d'informations au moment de l'inscription et à la fin de la période d'essaie. Les informations collectées sont :

- *Provenance* : le site internet visité juste avant `www.m2idapp.com`.
- *Emplacement :* le pays où réside l'utilisateur.
- *F.A.Q lue ? :* l'utilisateur doit dire si oui ou non il a lu la foire aux questions du site.
- *Pages vues :* le nombre de pages visitées sur `www.m2idapp.com`.

Le tableau suivant présente les données collectées auprès de 16 internautes :

Provenance	Emplacement	F.A.Q lue ?	Pages vues	Service choisi
Slashdot	USA	Oui	18	Aucun
Google	France	Oui	23	Premium
Digg	USA	Oui	24	Basique
Kiwitobes	France	Oui	23	Basique
Google	Angleterre	Non	21	Premium
(direct)	Allemagne	Non	12	Aucun
(direct)	Angleterre	Non	21	Basique
Google	USA	Non	24	Premium
Slashdot	France	Oui	19	Aucun
Digg	USA	Non	18	Aucun
Google	Angleterre	Non	18	Aucun
Kiwitobes	Angleterre	Non	19	Aucun
Digg	Allemagne	Oui	12	Basique
Slashdot	Angleterre	Non	21	Aucun
Google	Angleterre	Oui	18	Basique
Kiwitobes	France	Oui	19	Basique

La colonne “Service choisi” indique si l’utilisateur s’est abonné et si oui quel type d’abonnement il a souscrit. Le but de ce TP sera donc de prédire cette valeur à travers la construction d’un arbre de décision en python.

Les étapes qui suivent ne sont proposées qu’à titre indicatif (notamment celles relatives au choix des structures de données), l’étudiant ayant le loisir d’envisager toute autre manière de construire cet arbre de décision.

1. Créer une liste `mes_donnees` qui regroupe toutes les lignes du tableau ci-dessus sous la forme
   `mes_donnees=[['slashdot','USA','oui',18,'Aucun'],...]`

   Rajouter dans votre code la fonction `diviserjeu` qui divise les lignes en deux ensembles d’après les données d’une certaine colonne.

```
def diviserjeu(lignes,colonne,valeur):
  fonction_diviser=None
  if isinstance(valeur,int) or isinstance(valeur,float):
    fonction_diviser=lambda ligne:ligne[colonne]>=valeur
  else:
    fonction_diviser=lambda ligne:ligne[colonne]==valeur
  jeu1=[ligne for ligne in lignes if fonction_diviser(ligne)]
  jeu2=[ligne for ligne in lignes if not fonction_diviser(ligne)]
```

```
  return (jeu1,jeu2)
```

Cette fonction attend en arguments une liste de lignes (par exemple la liste `mes_donnees`), un numéro de colonnes (Ex : le numéro 2 représente la colonne "FAQ lue ?", sachant que la numérotation des listes commence par 0) et une valeur de séparation (Ex : les valeurs possibles pour la colonne 2 "FAQ lue ?" sont "Oui" et "Non"). La fonction `diviserjeu` retourne deux listes de lignes (deux jeux de données) : la première contiendra les lignes dont les valeurs de la colonne précisée correspondent à celle indiquée, la seconde contenant les autres.

Ainsi `diviserjeu(mes_donnees,2,'oui')` renvoie les deux listes `(jeu1,jeu2)` suivantes :

```
jeu1=[['slashdot', 'USA', 'oui', 18, 'Aucun'], ['google', 'France', 'oui', 23, 'Premium'],
['digg', 'USA', 'oui', 24, 'Basique'] , ['kiwitobes',
'France', 'oui', 23, 'Basique'], ['slashdot', 'France', 'oui',
19, 'Aucun'], ['digg', 'Allemagne', 'oui', 12, 'Basique'], ['google',
'Angleterre', 'oui', 18, 'Basique'], ['kiwitobes', 'France', 'oui', 19,
'Basique']]
```

```
jeu2=[['google', 'Angleterre', 'non', 21, 'Premium'], ['direct',
'New zeland', 'non', 12, 'Aucun'], ['direct', 'Angleterre', 'non', 21,
'Basique'], ['google', 'USA', 'non', 24, 'Premium'], ['digg',
'USA', 'non', 18, 'Aucun'], ['google', 'Angleterre', 'non', 18, 'Aucun'], ['kiwitobes',
'Angleterre', 'non', 19, 'Aucun'], ['slashdot',
'Angleterre', 'non', 21, 'Aucun']]
```

2. Pour choisir la meilleure division des lignes, on doit mesurer le niveau de mélange des listes retournées par `diviserjeu`. Cette mesure se fait à l'aide de *l'entropie* qui correspond à une mesure du désordre (ou mélange) d'un jeu de données. Ainsi, plus les groupes sont mélangés, plus leur entropie est élevée.

   Tout d'abord, nous allons rajouter la fonction `compteursuniques` qui dénombre chacun des résultats dans un jeu de données. `compteursuniques` crée des compteurs pour chaque résultat possible (la dernière colonne de chaque ligne contient le résultat), recherche tous les résultats possibles et les retourne dans un dictionnaire avec le nombre d'occurrences.

```
def compteursuniques(lignes):
  resultats={}
  for ligne in lignes:
    r=ligne[len(ligne)-1]
    if r not in resultats: resultats[r]=0
    resultats[r]+=1
  return resultats
```

Par exemple `compteursuniques(mes_donnees)` retourne le dictionnaire
`{'Aucun': 7, 'Premium': 3, 'Basique': 6}`

A partir de la fonction `compteursuniques`, construire la fonction `entropie` qui calcule l'entropie d'un jeu de donnée. On rappelle que l'entropie calcule la fréquence de chaque résultat possibles (son nombre d'occurrences divisé par le nombre total d'occurrences de tous les résultats) en utilisant la formule suivante :

$$entropie = \sum_{i \in A} -p(i) \times \log_2(p(i))$$

où A désigne l'ensemble de tous les résultats et $p(i)$ la fréquence du résultat i.

On trouvera par exemple :

```
>>> entropie(jeu1)
1.2987949407
```

3. A présent, nous allons créer la fonction `buildtree` qui construit récursivement un arbre de décision à partir d'un jeu de données. Analysez le code de cette fonction et testez la sur votre jeu de données.

```
def buildtree(rows,scoref=entropie):
   if len(rows)==0: return decisionnode()
   current_score=scoref(rows)

   # Variables de suivi du meilleur critère.
   best_gain=0.0
   best_criteria=None
   best_sets=None

   column_count=len(rows[0])-1
   for col in range(0,column_count):
     # Generer la liste des différentes valeurs de la colonne
     column_values={}
     for row in rows:
       column_values[row[col]]=1
     # Effectuer une division des lignes sur chaque valeur de la colonne.
     for value in column_values.keys():
       (set1,set2)=diviserjeu(rows,col,value)


       # Gain d'information
       p=float(len(set1))/len(rows)
       gain=current_score-p*scoref(set1)-(1-p)*scoref(set2)
       if gain>best_gain and len(set1)>0 and len(set2)>0:
         best_gain=gain
         best_criteria=(col,value)
         best_sets=(set1,set2)

   # Créer les sous branches
```

```
    if best_gain>0:
      trueBranch=buildtree(best_sets[0])
      falseBranch=buildtree(best_sets[1])
      return decisionnode(col=best_criteria[0],value=best_criteria[1],
                          tb=trueBranch,fb=falseBranch)

    else:
      return decisionnode(results=uniquecounts(rows))
```

4. La fonction suivante vous permet d'afficher votre arbre de décision :

```
def printtree(tree,indent=''):
  # Is this a leaf node?
  if tree.results!=None:
    print str(tree.results)

  else:
  # Print the criteria
    print str(tree.col)+':'+str(tree.value)+'? '

    # Print the branches
    print indent+'V->',
    printtree(tree.tb,indent+'  ')
    print indent+'F->',
    printtree(tree.fb,indent+'  ')
```

Analysez ce code et testez le avec vos données. Vous obtiendrez l'arbre de décision suivante :

```
>>> printtree(mes\_donnees,indent='')
0:google?
V-> 3:21?
  V-> {'Premium': 3}
  F-> 2:Oui?
    V-> {'Basique': 1}
    F-> {'Aucun': 1}
F-> 0:slashdot?
  V-> {'Aucun': 3}
  F-> 2:Oui?
    V-> {'Basique': 4}
    F-> 3:21?
      V-> {'Basique': 1}
      F-> {'Aucun': 3}
```

5. Analysez le code python suivant. Que fait-il ? Testez le avec vos données.

```
def classify(observation,tree):
  if tree.results!=None:
    return tree.results

  else:
    v=observation[tree.col]
    branch=None
    if isinstance(v,int) or isinstance(v,float):
      if v>=tree.value: branch=tree.tb
      else: branch=tree.fb
    else:
      if v==tree.value: branch=tree.tb
      else: branch=tree.fb
    return classify(observation,branch)
```