TP01 partie 2 : exploitation des données

Cette seconde partie est *optionnelle* pour le rendu de la mi-mars, **mais** sera très utile pour le rendu final. C'est bien joli de mettre en place des algorithmes mais cela n'a d'intérêt que si on les utilise sur des cas un peu conséquent.

Nous disposons de données sous format `.csv` et nous disposons de deux classes `Apriori` et `Arules`. C'est le résultat de `Arules.main` qui nous intéresse, par ailleurs la mise en place de `Arules` nécessite la sortie de `Apriori.main`. La chaîne de traitement est donc simple "data.csv → Apriori → Arules → exploitation"

Le passage du format de la donnée stockée au format de la donnée exploitée est une étape de *pré-processing*, le passage de la donnée calculée à la donnée obtenue en fin de traitement est une étape de *post-processing*

Un *itemset* dans le block **Apriori+Arules** est un tuple d'entiers, la donnée stockée et la donnée exploitée sont porteuses de *sens* pour l'humain, il est donc nécessaire de traduire l'information de (l'humain au programme ; puis du programme à l'humain)

> **Exemple**
>
> Dire la règle "(1,2) → 5" est prédictive n'a pas du tout le même effet
> que de dire "l'achat de {pain, beurre} est lié à l'achat de {lait}". Alors qu'il s'agit de la même information si l'on sait que **1** veut dire 'pain', **2** veut dire 'beurre' et **5** veut dire 'lait'

Les exemples dans le répertoire **data** vont nous permettre de voir ce dont on a besoin. L'objectif est d'obtenir à chaque fois un dictionnaire de la forme `{tid: liste, ...}` où `liste` est une liste d'entier > 0

Lorsque vous voulez ouvrir un fichier `csv` il est utile de regarder les premières lignes, afin de savoir si le fichier dispose d'une en-tête, s'il y a un index des valeurs. La commande pandas `read_csv` a de très nombreuses options, permettant de traiter les différentes situations

'sample_1.csv'

```
import pandas as pd
import numpy as np
base = 'data/'
df = pd.read_csv(base+'sample_1.csv')
print(df.head())
```

```
print(type(df['items'].loc[0]))
def str_list(ch:str) -> list:
    """ transforme '[1 , 2, 3]' en [1,2,3] """
    if ch.strip().startswith('[') and ch.strip().endswith(']'):
        _o = ch.replace('[','').replace(']','')
        return [int(x) for x in _o.split(',')]
    return ch
df['items'].apply(str_list)
print(type(df['items'].loc[0]))
df['items'] = df['items'].apply(str_list)
print(type(df['items'].loc[0]))
print(df.to_dict())
print(df['tid'].to_dict())
print(df['tid'].to_dict().values())
```

Ne reste plus qu'à écrire comment construire un dictionnaire comme on le souhaite

```
data = {x:v for x,v in zip(df['tid'].to_dict().values(),

df['items'].to_dict().values())}
```

'sample_2.csv'

La syntaxe est exactement la même dans ce fichier, nous allons utiliser une option, qui va nous permettre de lire et de transformer "à la volée" les informations lues. L'option `converters` permet de passer un dictionnaire de fonctions assurant le pré-traitement des valeurs récupérées

```
df = pd.read_csv(base+'sample_2.csv', converters={'itemset':str_list})
print(df.head())
data = {x:v for x,v in zip(df['tid'].to_dict().values(),
                           df['itemset'].to_dict().values())}
print("data", data)
algo = Apriori(data)
print("apriori with supp >= .5", algo.main(.5))
```

'sample_3.csv'

Ce fichier utilise un codage *one hot* on va utiliser cette propriété et un peu la bibliothèque `numpy` pour récupérer les informations qui nous intéresse

```
>>> df = pd.read_csv(base+'sample_3.csv')
>>> df.describe()
>>> df.values
>>> df.values[:,1:]
>>> df.values[:,1:] * np.array([1,2,3,4,5])
```

```
>>> [ [x for x in l if x>0] for l in df.values[:,1:] * np.array([1,2,3,4,5])
]
```

'sample_4csv'

Ce fichier utilise une description par chaîne de caractères, cette fois-ci on va, lire le fichier, découper le champ achats, construire une traduction numérique que l'on mettra de côté pour faire le traitement après le calcul des règles.

Les chaînes étant clairement entre guillemets, on va utiliser l'option `skipinitialspace` pour enlever les espaces.

```
df = pd.read_csv(base+'sample_4.csv', skipinitialspace=True)
df.head()
```

On aurait pu souhaiter découper une chaîne "pain beurre" et obtenir ['pain', 'beurre'] directement à la lecture, il suffit simplement de passer la découpe par l'option `converters`

```
>>> df = pd.read_csv("data/sample_4.csv", skipinitialspace=True, converters=
{"achats": lambda x: x.split(' ')})
>>> df.head()
```

L'étape suivante va consister à collecter tous les achats possibles et à produire un dictionnaire de traduction (dans les deux sens)

```
>>> df.achats.values
>>> sorted(set([x for l in df.achats.values for x in l]))
```

Une fois obtenue la liste, il est très simple d'obtenir 2 dictionnaires d'équivalence entre entier et mots:

```
def from_int_to_str(L:list) -> dict:
        """ given a list of str, provides a dictionnary int:str """
    return {i+1: v for i,v in enumerate(L)}
def from_str_to_int(L:list) -> dict:
        """ given a list of str, provides a dictionnary str:int """
        return {v:i+1 for i,v in enumerate(L)}
```

Et construire la nouvelle table :

```
values = sorted(set([x for l in df.achats.values for x in l]))
_1 = from_int_to_str(values)
_2 = from_str_to_int(values)

df['itemsets'] = [[_2[x] for x in l] for l in df.achats.values]
print(df.head())
```

Et voilà ...

```
   tid                                achats         itemsets
0    1                  [lait, pain, fruits]        [5, 7, 4]
1    2               [beurre, oeufs, fruits]        [1, 6, 4]
2    3                     [bieres, couches]           [2, 3]
3    4  [lait, pain, beurre, oeufs, fruits]  [5, 7, 1, 6, 4]
4    5                                [pain]              [7]
```

Ne reste plus qu'à exploiter

```
data = {x:v for x,v in zip(df['tid'].to_dict().values(),
                           df['itemsets'].to_dict().values())}
print("data", data)
algo = Apriori(data)
rules = Arules(algo.main(.25), algo.support_history)
df_rules = rules.main(.5)
print(df_rules.describe())
```

Dont la sortie donne

```
       lhs_support  rhs_support    support  confidence       lift   leverage
conviction
count    27.000000    27.000000  27.000000   27.000000  27.000000  27.000000
27.000000
mean      0.476190     0.423280   0.306878    0.666667   1.685185   0.104308
inf
std       0.088596     0.108374   0.051716    0.165056   0.655523   0.069860
NaN
min       0.285714     0.285714   0.285714    0.500000   0.875000  -0.040816
0.857143
25%       0.428571     0.285714   0.285714    0.500000   1.166667   0.040816
1.285714
50%       0.428571     0.428571   0.285714    0.666667   1.555556   0.102041
1.714286
75%       0.571429     0.500000   0.285714    0.666667   1.750000   0.163265
2.142857
max       0.571429     0.571429   0.428571    1.000000   3.500000   0.204082
inf
```

On peut reconstruire la règle en utilisant le dictionnaire inverse

```
df_rules['rule'] = ["{} -> {}".format(a,b)
                    for a,b in zip([[_1[x] for x in l]
                                    for l in df_rules.lhs.values],
                                   [[_1[x] for x in l]
                                    for l in df_rules.rhs.values])]
```

```
print(df_rules.sort_values(by=['confidence', 'lift', 'leverage'],
                                                    ascending=False).head())
```

```
     lhs   rhs  lhs_support  rhs_support   support  confidence  lift
leverage  conviction                        rule
8   (3,)  (2,)     0.285714     0.285714  0.285714        1.00  3.50
0.204082         inf  ['couches'] -> ['bieres']
9   (2,)  (3,)     0.285714     0.285714  0.285714        1.00  3.50
0.204082         inf  ['bieres'] -> ['couches']
4   (6,)  (1,)     0.428571     0.571429  0.428571        1.00  1.75
0.183673         inf    ['oeufs'] -> ['beurre']
17  (5,)  (7,)     0.428571     0.571429  0.428571        1.00  1.75
0.183673         inf       ['lait'] -> ['pain']
5   (1,)  (6,)     0.571429     0.428571  0.428571        0.75  1.75
0.183673    2.285714    ['beurre'] -> ['oeufs']
```