1 Objectif

Extraction d'itemsets fréquents et de règles d'association sous Python. Utilisation de la librairie ''mlxtend''.

Ça fait un moment que je n'ai plus écrit de didacticiels sur les règles d'association. Comme je m'investis de plus en plus dans Python dans mes enseignements, je me suis dit qu'il était temps d'en écrire un pour ce langage, en complément d'un ancien document consacré à différents logiciels, notamment R (''Règles d'association – Comparaison de logiciels'', novembre 2008). J'utilise la librairie ''mlxtend'' (machine learning extensions) qui propose une série d'outils pour le machine learning : clustering, classification supervisée, régression, etc., et donc l'extraction des itemsets fréquents et des règles d'association que nous étudierons dans ce didacticiel.

L'organisation de ce document est on ne peut plus classique dans notre contexte : chargement et préparation des données, extraction des itemsets fréquents, recherche des sous-ensembles d'itemsets comportant des items particuliers, déduction des règles d'association à partir des itemsets fréquents, recherche de sous-ensembles de règles au regard de la présence de certains items ou répondant à des critères numériques.

2 Importation et préparation des données

2.1 Données transactionnelles

Le fichier ''**market_basket.txt**'' est un fichier texte qui se présente sous la forme d'une base transactionnelle (voir – ''Règles d'association – Données transactionnelles'', décembre 2010 – pour les spécificités de cette organisation des données). Nous avons un tableau de données sur deux colonnes : la première correspond aux identifiants de transaction, la seconde aux noms des produits. Voici les premières lignes du fichier :

ID	Product
1	Peaches
2	Vegetable_Oil
2	Frozen_Corn
3	Plums
4	Pancake_Mix
5	Cheese
6	Cauliflower
7	2pct_Milk
8	98pct_Fat_Free_Hamburger
8	Potato_Chips
8	Sesame_Oil
8	Ice_Cream_Sandwich

Si les transactions représentent des caddies de supermarché : nous avons le seul produit {Peaches} dans le 1er caddie ; deux produits {Vegetable_Oil, Frozen_Com} dans le 2nd ; etc. Ainsi, l'identifiant d'une transaction apparaîtra autant de fois qu'il y a de produits dans le caddie associé, et un produit peut bien sûr apparaître dans plusieurs caddies.

Nous importons les données en faisant appel à la librairie ''pandas''. Nous affichons les dix premières lignes.

```
#changement de dossier
import os
os.chdir("… votre dossier de travail …")

#importation des données
import pandas
D = pandas.read_table("market_basket.txt",delimiter="\t",header=0)

#10 premières lignes
print(D.head(10))
```

```
   ID                  Product
0   1                  Peaches
1   2            Vegetable_Oil
2   2              Frozen_Corn
3   3                    Plums
4   4              Pancake_Mix
5   5                   Cheese
6   6              Cauliflower
7   7                2pct_Milk
8   8  98pct_Fat_Free_Hamburger
9   8             Potato_Chips
```

Nous vérifions les dimensions de la table.

```
#vérification des dimensions
print(D.shape)
```

```
(12935, 2)
```

2.2 Transformation en tableau binaire

La librairie que nous utiliserons prend en entrée un tableau binaire ''transactions x produits'' où sont recensés la présence (codé 1 ou True) des produits dans chaque caddie (l'absence est codée 0 ou False). Un tableau croisant ID et PRODUCT fait parfaitement l'affaire. Nous affichons les 20 premières transactions et les 3 premiers produits.

```
#tableau croisé 0/1
TC = pandas.crosstab(D.ID,D.Product)
print(TC.iloc[:20,:3])
```

Product	100_Watt_Lightbulb	2pct_Milk	40_Watt_Lightbulb
ID			
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	1	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0
16	0	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0

Nous distinguons par exemple la présence du produit ‘’2pct_Milk’’ dans la transaction n°7.

La dimension du tableau TC est de (1360, 303) c.-à-d. notre fichier recense 1360 transactions et 303 produits.

```
#dimensions
print(TC.shape)
```

```
(1360, 303)
```

Nous pouvons obtenir la liste des produits en affichant les en-têtes de colonnes du tableau TC.

```
#liste des noms de produits
print(TC.columns)
```

```
Index(['100_Watt_Lightbulb', '2pct_Milk', '40_Watt_Lightbulb',
       '60_Watt_Lightbulb', '75_Watt_Lightbulb', '98pct_Fat_Free_Hamburger',
       'AA_Cell_Batteries', 'Apple_Cinnamon_Waffles', 'Apple_Drink',
       'Apple_Fruit_Roll',
       ...
       'White_Bread', 'White_Wine', 'White_Zinfandel_Wine', 'Whole_Corn',
       'Whole_Green_Beans', 'Whole_Milk', 'Window_Cleaner', 'Wood_Polish',
       'flav_Fruit_Bars', 'flav_Ice'],
      dtype='object', name='Product', length=303)
```

Nous travaillons à partir de ce tableau binaire dorénavant.

3 Extraction des itemsets fréquents

Nous utilisons la librairie ''mlxtend''. Nous devons l'installer lors de la première utilisation. La démarche est décrite sur le site de l'éditeur (http://rasbt.github.io/mlxtend/installation/). Elle a fonctionné sans embûches en ce qui me concerne (Python 3.7.1, version 0.13.0 de ''mlxtend'').

3.1 Extraction des itemsets

Nous importons la fonction apriori() et nous l'appliquons à nos données en fixant un support minimum de 0.025 c.-à-d. nous acceptons les itemsets avec un support (min_support=0.025) supérieur ou égal à 0.025 x 1360 = 34 transactions, un cardinal inférieur ou égal à 4 (max_len=4), et nous demandons à ce que les noms des items apparaissent explicitement à l'affichage (use_colnames=True).

```
#importation de la fonction apriori
from mlxtend.frequent_patterns import apriori

#itemsets frequents
freq_itemsets = apriori(TC,min_support=0.025,max_len=4,use_colnames=True)
```

Les résultats sont stockés dans une structure de type ''pandas / DataFrame''.

```
#type -> pandas DataFrame
type(freq_itemsets)
```

```
pandas.core.frame.DataFrame
```

Elle est composée de 2 colonnes : le support et la description des itemsets.

```
#liste des colonnes
print(freq_itemsets.columns)
```

```
Index(['support', 'itemsets'], dtype='object')
```

Pour nos données et avec les paramètres ci-dessus, nous avons obtenus 603 itemsets fréquents.

```
#nombre d'itemsets
print(freq_itemsets.shape)
```

```
(603, 2)
```

Affichons les 15 premiers :

```
#affichage des 15 premiers itemsets
print(freq_itemsets.head(15))
```

```
     support                  itemsets
0   0.030147       (100_Watt_Lightbulb)
1   0.109559                (2pct_Milk)
2   0.037500        (60_Watt_Lightbulb)
3   0.031618        (75_Watt_Lightbulb)
```

```
4   0.093382  (98pct_Fat_Free_Hamburger)
5   0.031618         (AA_Cell_Batteries)
6   0.025735    (Apple_Cinnamon_Waffles)
7   0.026471               (Apple_Drink)
8   0.031618          (Apple_Fruit_Roll)
9   0.032353                 (Apple_Jam)
10  0.033088               (Apple_Jelly)
11  0.032353               (Apple_Sauce)
12  0.053676                    (Apples)
13  0.066912                   (Aspirin)
14  0.027941               (Avocado_Dip)
```

3.2 Filtrage des itemsets

Nous souhaitons pouvoir filtrer les itemsets selon la présence d'items en particulier (ex. quels sont les itemsets qui contiennent le produit ''Aspirin'' ?). Pour ce faire, nous devons déjà comprendre la structure de la colonne ''itemsets'' du DataFrame. Il s'agit d'une ''pandas. Series''

```
#type du champ 'itemsets'
print(type(freq_itemsets.itemsets))
```

```
<class 'pandas.core.series.Series'>
```

Et pour ce qui est de chaque élément ? Accédons au premier itemset par exemple.

```
#accès indexé au premier élément
print(freq_itemsets.itemsets[0])
```

```
frozenset({'100_Watt_Lightbulb'})
```

Un itemset correspond à un objet frozenset de Python, un type ensemble (set) qui n'est pas modifiable (on dit non-mutable). Plusieurs solutions s'offrent à nous pour la recherche d'itemsets répondant à des conditions de présence d'items.

3.2.1 Solution 1 – Utilisation de la méthode apply() de pandas.Series

La première consiste à exploiter la méthode apply() de l'objet pandas.Series. Elle permet d'appliquer une fonction sur chaque élément de la série de valeurs, un peu à la manière des fonction apply() de R si on essaie de faire un parallèle entre les deux outils.

D'abord, nous écrivons une fonction is_inclus() qui permet de vérifier si un sous-ensemble items est inclus dans l'ensemble x. La fonction renvoie True dans ce cas.

```
#fonction de test d'inclusion
def is_inclus(x,items):
    return items.issubset(x)
```

Nous appliquons cette fonction sur chaque élément de la colonne ‘’itemsets’’ recensant les itemsets fréquents. Nous obtenons les numéros des itemsets concernés.

```
#recherche des index des itemsets correspondant à une condition
import numpy
id = numpy.where(freq_itemsets.itemsets.apply(is_inclus,items={'Aspirin'}))
print(id)
```

```
(array([ 13, 208, 249, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281,
       282, 283, 284, 285, 286, 509, 510, 511, 552, 553, 554, 555, 556],
      dtype=int64),)
```

Les itemsets (13, 208, 249, etc.) comprennent le produit ‘Aspirin’. Nous pouvons les afficher explicitement.

```
#affichage des itemsets corresp.
print(freq_itemsets.loc[id])
```

```
      support                              itemsets
13   0.066912                             (Aspirin)
208  0.034559                 (Aspirin, 2pct_Milk)
249  0.027941   (Aspirin, 98pct_Fat_Free_Hamburger)
272  0.027206                      (Cola, Aspirin)
273  0.025735             (Domestic_Beer, Aspirin)
274  0.038235                       (Eggs, Aspirin)
275  0.027206                  (Aspirin, Hot_Dogs)
276  0.027206                    (Aspirin, Onions)
277  0.027206   (Pepperoni_Pizza_-_Frozen, Aspirin)
278  0.025000              (Popcorn_Salt, Aspirin)
279  0.036029              (Aspirin, Potato_Chips)
280  0.030147                  (Aspirin, Potatoes)
281  0.030147              (Aspirin, Sweet_Relish)
282  0.028676              (Toilet_Paper, Aspirin)
283  0.025000                  (Tomatoes, Aspirin)
284  0.030882                (Aspirin, Toothpaste)
285  0.025000               (Wheat_Bread, Aspirin)
286  0.041912               (White_Bread, Aspirin)
509  0.025735           (Eggs, Aspirin, 2pct_Milk)
510  0.025000    (Aspirin, Potato_Chips, 2pct_Milk)
511  0.027206     (White_Bread, Aspirin, 2pct_Milk)
552  0.025000         (Eggs, Aspirin, Potato_Chips)
553  0.029412          (Eggs, White_Bread, Aspirin)
554  0.027206  (White_Bread, Aspirin, Potato_Chips)
555  0.025000      (White_Bread, Aspirin, Potatoes)
556  0.025735    (White_Bread, Aspirin, Toothpaste)
```

3.2.2 Solution 1’ – Utilisation de la fonction anonyme lambda

Pour les impatients, nous pouvons définir à la volée la fonction de comparaison lors de l’appel de la méthode apply() de Series via le mécanisme des fonctions anonymes lambda de Python.

```
#passer par une fonction lambda si on est pressé
numpy.where(freq_itemsets.itemsets.apply(lambda x,ensemble:ensemble.issubset(x),ensemble={'Aspirin'}))
```

```
(array([ 13, 208, 249, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281,
        282, 283, 284, 285, 286, 509, 510, 511, 552, 553, 554, 555, 556],
       dtype=int64),)
```

La liste des numéros d'itemsets est la même bien évidemment mais, ici, une seule de commande permet de produire le résultat attendu.

3.2.3 Solution 2 – Utilisation des opérateurs de comparaison de pandas.Series

En explorant un peu plus la documentation de la classe pandas.Series, je me suis rendu compte qu'elle proposait des fonctions de comparaison vectorisées (Binary operators functions - https://pandas.pydata.org/pandas-docs/stable/reference/series.html). Nous pouvons exploiter cette fonctionnalité. Pour obtenir les itemsets comprenant le produit 'Aspirin' par exemple :

```
#itemsets contenant Aspirin - passer par les méthodes natives de Series
print(freq_itemsets[freq_itemsets['itemsets'].ge({'Aspirin'})])
```

```
      support                                itemsets
13   0.066912                                (Aspirin)
208  0.034559                   (Aspirin, 2pct_Milk)
249  0.027941   (Aspirin, 98pct_Fat_Free_Hamburger)
272  0.027206                       (Cola, Aspirin)
273  0.025735              (Domestic_Beer, Aspirin)
274  0.038235                       (Eggs, Aspirin)
275  0.027206                    (Aspirin, Hot_Dogs)
276  0.027206                      (Aspirin, Onions)
277  0.027206   (Pepperoni_Pizza_-_Frozen, Aspirin)
278  0.025000                (Popcorn_Salt, Aspirin)
279  0.036029                (Aspirin, Potato_Chips)
280  0.030147                    (Aspirin, Potatoes)
281  0.030147                (Aspirin, Sweet_Relish)
282  0.028676                (Toilet_Paper, Aspirin)
283  0.025000                    (Tomatoes, Aspirin)
284  0.030882                  (Aspirin, Toothpaste)
285  0.025000                 (Wheat_Bread, Aspirin)
286  0.041912                 (White_Bread, Aspirin)
509  0.025735            (Eggs, Aspirin, 2pct_Milk)
510  0.025000    (Aspirin, Potato_Chips, 2pct_Milk)
511  0.027206     (White_Bread, Aspirin, 2pct_Milk)
552  0.025000         (Eggs, Aspirin, Potato_Chips)
553  0.029412          (Eggs, White_Bread, Aspirin)
554  0.027206  (White_Bread, Aspirin, Potato_Chips)
555  0.025000      (White_Bread, Aspirin, Potatoes)
556  0.025735    (White_Bread, Aspirin, Toothpaste)
```

Nous obtenons le bon résultat, de manière autrement plus simple finalement. Mais bon, apprendre à exploiter efficacement la méthode apply() constitue un bon investissement je pense ;

de même, j'ai trouvé intéressant de montrer l'utilisation des fonctions anonymes lambda sous Python.

Essayons maintenant quelques requêtes avec le système des comparaisons.

- L'itemset contenant exclusivement l'item 'Aspirin'.

```
#itemset avec Aspirin
print(freq_itemsets[freq_itemsets['itemsets'].eq({'Aspirin'})])
```

```
     support   itemsets
13  0.066912  (Aspirin)
```

- Les itemsets contenant les produits 'Aspirin' et 'Eggs' (je ne vois pas le rapport mais pourquoi pas après tout ?).

```
#itemsets contenant Aspirin et Eggs
print(freq_itemsets[freq_itemsets['itemsets'].ge({'Aspirin','Eggs'})])
```

```
      support                      itemsets
274  0.038235               (Eggs, Aspirin)
509  0.025735     (Eggs, Aspirin, 2pct_Milk)
552  0.025000  (Eggs, Aspirin, Potato_Chips)
553  0.029412   (Eggs, White_Bread, Aspirin)
```

- La position des items dans la requête n'influe pas sur les résultats bien sûr.

```
#itemsets contenant Aspirin et Eggs
print(freq_itemsets[freq_itemsets['itemsets'].ge({'Eggs','Aspirin'})])
```

```
      support                      itemsets
274  0.038235               (Eggs, Aspirin)
509  0.025735     (Eggs, Aspirin, 2pct_Milk)
552  0.025000  (Eggs, Aspirin, Potato_Chips)
553  0.029412   (Eggs, White_Bread, Aspirin)
```

4 Extraction (déduction) des règles d'association

4.1 Génération des règles

Nous produisons les règles à partir des itemsets fréquents. Elles peuvent être très nombreuses, nous en limitons la prolifération en définissant un seuil minimal (min_threshold = 0.75) sur une mesure d'intérêt, en l'occurrence la confiance dans notre exemple (metric = ''confidence'').

```
#fonction de calcul des règles
from mlxtend.frequent_patterns import association_rules

#génération des règles à partir des itemsets fréquents
regles = association_rules(freq_itemsets,metric="confidence",min_threshold=0.75)
```

```
#type de l'objet renvoyé
print(type(regles))
```

```
<class 'pandas.core.frame.DataFrame'>
```

La fonction association_rules() renvoie un objet de type pandas.DataFrame.

Nous disposons de 50 règles, décrites par 9 caractéristiques…

```
#dimension
print(regles.shape)
```

```
(50, 9)
```

… que sont : l'antécédent, le conséquent, et 7 indicateurs numériques d'évaluation des règles.

```
#liste des colonnes
print(regles.columns)
```

```
Index(['antecedents', 'consequents', 'antecedent support',
       'consequent support', 'support', 'confidence', 'lift', 'leverage',
       'conviction'],
      dtype='object')
```

Voici par exemple les 5 « premières » règles :

```
#5 "premières" règles
print(regles.iloc[:5,:])
```

```
                 antecedents    consequents     ...      leverage  conviction
0       (Aspirin, 2pct_Milk)  (White_Bread)     ...      0.023089    4.140147
1       (Bananas, 2pct_Milk)  (White_Bread)     ...      0.021969    4.734743
2     (Bananas, White_Bread)    (2pct_Milk)     ...      0.022191    4.353268
3        (Cola, Wheat_Bread)    (2pct_Milk)     ...      0.022191    4.353268
4  (Popcorn_Salt, 2pct_Milk)         (Eggs)     ...      0.023143    4.934283
```

Il faudrait adapter l'affichage pour disposer de toutes les informations. Dans ce qui suit, nous nous en tiendrons à l'antécédent, le conséquent et le lift. Nous modifions aussi le format d'affichage. Voici de nouveau les 5 « premières » règles.

```
#règles en restreignant l'affichage à qqs colonnes
myRegles = regles.loc[:,['antecedents','consequents','lift']]
print(myRegles.shape)

#pour afficher toutes les colonnes
pandas.set_option('display.max_columns',5)
pandas.set_option('precision',3)

#affichage des 5 premières règles
print(myRegles[:5])
```

```
                 antecedents    consequents   lift
0       (Aspirin, 2pct_Milk)  (White_Bread)  6.609
1       (Bananas, 2pct_Milk)  (White_Bread)  6.833
2     (Bananas, White_Bread)    (2pct_Milk)  7.261
3        (Cola, Wheat_Bread)    (2pct_Milk)  7.261
4  (Popcorn_Salt, 2pct_Milk)         (Eggs)  6.696
```

4.2 Filtrage des règles

Nous pouvons filtrer ou organiser les règles de différentes manières.

- Afficher les règles qui présentent un LIFT supérieur ou égal à 7

```
#affichage des règles avec un LIFT supérieur ou égal à 7
print(myRegles[myRegles['lift'].ge(7.0)])
```

```
                              antecedents                 consequents   lift
2                    (Bananas, White_Bread)                (2pct_Milk)  7.261
3                       (Cola, Wheat_Bread)                (2pct_Milk)  7.261
8                     (Wheat_Bread, Onions)                (2pct_Milk)  7.574
10                  (Potatoes, Wheat_Bread)                (2pct_Milk)  7.053
13                (Wheat_Bread, Toothpaste)                (2pct_Milk)  7.380
16            (White_Bread, Hamburger_Buns)  (98pct_Fat_Free_Hamburger)  8.202
17  (Wheat_Bread, 98pct_Fat_Free_Hamburger)               (White_Bread)  7.556
29             (Hot_Dog_Buns, Sweet_Relish)                 (Hot_Dogs)  9.031
35                 (Potatoes, Toilet_Paper)               (White_Bread)  7.319
37               (Toilet_Paper, Toothpaste)               (White_Bread)  7.346
41        (Eggs, Potato_Chips, White_Bread)                (2pct_Milk)  7.143
44          (Eggs, Toothpaste, White_Bread)                (2pct_Milk)  7.261
46     (White_Bread, 2pct_Milk, Toothpaste)              (Potato_Chips)  7.726
47   (White_Bread, 2pct_Milk, Potato_Chips)               (Toothpaste)  9.514
48  (White_Bread, Toothpaste, Potato_Chips)                (2pct_Milk)  7.569
49    (2pct_Milk, Toothpaste, Potato_Chips)               (White_Bread)  7.319
```

- Les 10 règles qui présentent les LIFT les plus élevés. Nous passons par un tri décroissant de la base de règles selon le LIFT.

```
#trier les règles dans l'ordre du lift décroissants - 10 meilleurs règles
print(myRegles.sort_values(by='lift',ascending=False)[:10])
```

```
                              antecedents                 consequents   lift
47   (White_Bread, 2pct_Milk, Potato_Chips)               (Toothpaste)  9.514
29             (Hot_Dog_Buns, Sweet_Relish)                 (Hot_Dogs)  9.031
16            (White_Bread, Hamburger_Buns)  (98pct_Fat_Free_Hamburger)  8.202
46     (White_Bread, 2pct_Milk, Toothpaste)              (Potato_Chips)  7.726
8                     (Wheat_Bread, Onions)                (2pct_Milk)  7.574
48  (White_Bread, Toothpaste, Potato_Chips)                (2pct_Milk)  7.569
17  (Wheat_Bread, 98pct_Fat_Free_Hamburger)               (White_Bread)  7.556
13                (Wheat_Bread, Toothpaste)                (2pct_Milk)  7.380
37               (Toilet_Paper, Toothpaste)               (White_Bread)  7.346
49    (2pct_Milk, Toothpaste, Potato_Chips)               (White_Bread)  7.319
```

– Liste des règles menant à la conclusion {‘2pct_Milk’}

```
#filtrer les règles menant au conséquent {‘2pct_milk’}
print(myRegles[myRegles['consequents'].eq({'2pct_Milk'})])
```

```
                                antecedents  consequents   lift
2                     (Bananas, White_Bread)  (2pct_Milk)  7.261
3                        (Cola, Wheat_Bread)  (2pct_Milk)  7.261
7                        (Eggs, Wheat_Bread)  (2pct_Milk)  6.970
8                      (Wheat_Bread, Onions)  (2pct_Milk)  7.574
9                  (Toothpaste, Potato_Chips)  (2pct_Milk)  6.980
10                    (Potatoes, Wheat_Bread)  (2pct_Milk)  7.053
13                  (Wheat_Bread, Toothpaste)  (2pct_Milk)  7.380
41         (Eggs, Potato_Chips, White_Bread)  (2pct_Milk)  7.143
44           (Eggs, Toothpaste, White_Bread)  (2pct_Milk)  7.261
48  (White_Bread, Toothpaste, Potato_Chips)  (2pct_Milk)  7.569
```

– Liste des règles dont l’antécédent comporte l’item ‘Aspirin’.

```
#filtrer les règles contenant 'Aspirin' dans leur antécédent
print(myRegles[myRegles['antecedents'].ge({'Aspirin'})])
```

```
                antecedents    consequents   lift
0       (Aspirin, 2pct_Milk)  (White_Bread)  6.609
18           (Eggs, Aspirin)  (White_Bread)  6.458
19  (Aspirin, Potato_Chips)  (White_Bread)  6.339
20       (Potatoes, Aspirin)  (White_Bread)  6.962
21     (Aspirin, Toothpaste)  (White_Bread)  6.996
```

5 Conclusion

Nous avons montré dans ce tutoriel qu’il est relativement simple de générer et manipuler les règles d’association sous Python via la librairie ‘’mlxtend’’.

En parcourant la documentation, je me suis rendu compte que la librairie allait au-delà de ces approches et proposait des solutions étendues pour la data science : algorithmes de machine learning, sélection et construction de variables, traitement d’images, text mining, graphiques, etc. Elle semble constituer – il me semble, il faut explorer dans le détail pour se faire une idée précise – une alternative ou un complément crédible au très populaire ‘’scikit-learn’’.

6 Références

‘’Mlxtend : machine learning extensions’’, http://rasbt.github.io/mlxtend/

Tutoriel Tanagra, ‘’Extraction de règles d’association – Diapos’’, Juin 2015.