TP01 les règles d'association

La classe `Arules` (fichier rules.py)

L'objectif de cette classe est de construire, à partir du calcul effectué par la classe `Apriori` les règles les plus pertinentes. Une règle est constituée d'une partie gauche (abrégée en **lhs** pour '*left hand side*') et d'une partie droite (abrégée en **rhs** pour '*right hand side*'). Intuitivement avec un itemset de taille 3 `(a, b, c)` on peut constituer les règles suivantes <lhs ; rhs>

- <a,b ; c> ; <a,c ; b> ; <b,c ; a>
- <a ; b,c> ; <b; a,c> ; <c; a,b>
  Certaines de ces règles sont pertinentes, d'autres pas. Pour décider de cette pertinence on attribue différentes mesures.

constructeur

Reçoit en entrée 2 informations une liste de listes d'itemsets, un dictionnaire dont les clefs sont les itemsets et les valeurs sont les supports associés. Ces deux informations sont stockées dans deux attributs `self.list_itemsets` et `self.support_itemsets`. Puis le constructeur appelle la méthode `reset`

reset

Méthode sans paramètre, ne renvoie rien. Elle se contente d'initialiser l'attribut `self.rules` comme une liste vide.

```
ar = Arules([[(1,), (2,), (3,), (4,)], [ (1,2), (1,4), (2,3)]],
                        {(1,): .5, (2,): .5, (3,): .5, (4,): .5,
                        (1,2): .3, (1,4): .25, (2,3): .4})
print("items", ar.list_itemsets)
print("support", ar.support_itemsets)
print("regles", ar.rules)
print("{0} Exemple 2 {0}".format("="*7))
data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
db = Apriori(data)
br = Arules(db.main(.5), db.support_history)
print("items", br.list_itemsets)
print("support", br.support_itemsets)
print("regles", br.rules)
```

Et voici ce que l'on obtient

```
items [[(1,), (2,), (3,), (4,)], [(1, 2), (1, 4), (2, 3)]]
support {(1,): 0.5, (2,): 0.5, (3,): 0.5, (4,): 0.5, (1, 2): 0.3, (1, 4):
0.25, (2, 3): 0.4}
regles []
======= Exemple 2 =======
items [[(1,), (2,), (3,), (5,)], [(1, 3), (2, 3), (2, 5), (3, 5)], [(2, 3,
5)]]
support {(1,): 0.5, (3,): 0.75, (2,): 0.75, (5,): 0.75, (1, 3): 0.5, (2, 3):
0.5, (2, 5): 0.75, (3, 5): 0.5, (2, 3, 5): 0.5}
regles []
```

métriques

Nous allons mettre en place les métriques présentées en cours : le *support*, la *confiance*, l'*amélioration* le *levier* et la *conviction*. La signature sera toujours la même : $\text{tuple} \times \text{tuple} \rightarrow float$. Le premier tuple correspond à l'itemset en partie gauche de la règle, le second l'itemset en partie droite et le résultat sera calculé en accord avec le support de cours. Le nom des méthodes sera le nom anglophone : **support, confidence, lift, leverage, conviction**

Attention la conviction utilise au dénominateur (1 - confiance), lorsque la confiance vaudra 1, on renverra `None`

En plus de la méthode `lift`, on va créer la méthode `lift_diag` qui prendra en entrée 2 tuples et renverra en sortie un message (une chaîne de caractères) en fonction du calcul fait par la méthode `lift`

- si `lift(lhs, rhs)` vaut 1 le message sera "ne pas utiliser lhs → rhs"
- si `lift(lhs, rhs)` < 1 le message sera "lhs et rhs ne peuvent pas co-exister dans une règle"
- si `lift(lhs, rhs)` > 1 le message sera "lhs → rhs est prédictive"

```
data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
db = Apriori(data)
br = Arules(db.main(.5), db.support_history)
print("#=== Evaluation des règles à partir du triplet (2,3,5) ===#")
for lhs,rhs in [ [(2,5), (3,)], [(2,3), (5,)], [(3,5), (2,)] ]:
    print("evaluation de {} -> {}".format(lhs, rhs))
    for m in "support confidence conviction leverage lift
lift_diag".split():
            _r = getattr(br, m)(lhs, rhs)
            _r = round(_r, 3) if isinstance(_r, float) else _r
            print("{} : {}".format(getattr(br, m).__name__, _r))

    print("#"+"="*73+"#")
```

On obtient la sortie suivante

```
#=== Evaluation des règles à partir du triplet (2,3,5) ===#
evaluation de (2, 5) -> (3,)
support : 0.5
confidence : 0.667
conviction : 0.751
leverage : -0.062
lift : 0.889
lift_diag : (2, 5) et (3,) ne peuvent pas co-exister dans une règle
#=========================================================================#
evaluation de (2, 3) -> (5,)
support : 0.5
confidence : 1.0
conviction : None
leverage : 0.125
lift : 1.333
lift_diag : (2, 3) -> (5,) est prédictive
#=========================================================================#
evaluation de (3, 5) -> (2,)
support : 0.5
confidence : 1.0
conviction : None
leverage : 0.125
lift : 1.333
lift_diag : (3, 5) -> (2,) est prédictive
#=========================================================================#
```

Description de l'algorithme http://rakesh.agrawal-family.com/papers/vldb94apriori_rj.pdf page 14

Attention les algorithmes décrits n'ont pas la signature exacte des méthodes que nous allons construire. Ces algorithmes sont donnés en programmation impérative, pas en objet

```
def generate_rules(minconf:float):
        """ parcourt les k-itemsets pour produire les règles
            minconf est la confiance minimale pour accepter une règle
        """
        rules = vide
        forall l_k un k-itemsets, k>1 faire
                Construire RHS_1 = { 1-itemset appartenant à l_k }
                Si k==2: Ajouter validation_rules(lk, RHS_1, minconf) dans
rules
                Sinon Ajouter build_rules(l_k, RHS_1, minconf) dans rules
                fsi
        fait
```

```
def build_rules(lk: k-itemsets, RHS_p: p-itemsets en partie droite,
seuil:float):
        """ fonction récursive """
        local = vide
        Si k > p+1:
                RHS_q = cross_product(RHS_p, p+1)
                Ajouter validation_rules(lk, RHS_q, seuil) dans local
                Ajouter build_rules(lk, RHS_q, seuil) dans local
    fsi
    renvoyer local
```

```
def cross_product(L:liste of k-itemsets, k+1):
    """ à partir des éléments de L on construit les nouveaux éléments
        rappel un itemset est un tuple
        on va construire à partir de 2 tuples un nouveau tuple
        sous certaines conditions
        exple L = [(1,2), (1,3), (2,3), (2,4)]
        on renverra [ (1,2,3) ]
        car (1,2), (1,3) et (2,3) sont dans L
        mais il n'y aura pas (2,3,4) car (3,4) n'est pas dans L
    """
    rep = []
    taille = len(L)
    pour i in range(taille -1)
                j = j+1
                tant que j < taille and les k-1 premières valeurs de
L[i]==L[j] faire
                        construire nouveau à partir de L[i] et du dernier de
L[j]
                        # nouveau est de taille k+1
                        si tous les sous-itemsets de taille k de nouveau
sont dans L[i]
                                ajouter nouveau dans rep
                        fsi
                        j = j+1
            fait
        fpour
        return rep
```

```
def validation_rules(lk: k-itemset, RHS: liste de p-itemsets, seuil:float):
        """ p < k et chaque élément de RHS et une partie de lk
            on va construire pour chaque p-itemset rhs de RHS
            la règle lk - rhs -> rhs
            on va calculer la confiance de la règle et la garder si
            cette confiance est >= seuil

                astuce: on a 2 itemsets représentés par des tuples triés
                        pour construire lhs, on va passer aux ensembles
```

```
                    et reconstruire un tuple trié
            >>> lk = (1,2,3,4) ; rhs = (2,4)
            >>> set(lk)
            {1, 2, 3, 4}
            >>> set(rhs)
            {2, 4}
            >>> set(lk) - set(rhs)
            {1, 3}
            >>> sorted({31,4})
            [4, 31]
            >>> tuple(sorted({3,-1,42,17}))
            (-1, 3, 17, 42)
        """
```

Réalisation

L'algorithme a été présenté de manière *descendante* (en anglais *top down*), alors que nous allons faire une construction *ascendante* (en anglais *bottom up*)

`cross_product`

Reçoit en entrée une liste d'*itemsets* et la taille des éléments de la liste, elle renvoie une liste d'itemsets de taille+1

Vous avez un exemple de sortie attendue dans le docstring qui présente l'algorithme

`validation_rules`

Reçoit en entrée un *itemset*, une liste d'*itemsets*, un réel dans $[0, 1]$ et renvoie une sous-liste des *itemsets* qui ont été acceptés, de plus, elle met à jour l'attribut `rules`. Pour être accepté il faut que l'indice de *confiance* de la règle générée soit supérieur ou égal au seuil fixé. Par ailleurs les règles seront enregistrées sous la forme `(partie-gauche, partie-droite)`

```
    data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
    db = Apriori(data)
    br = Arules(db.main(.5), db.support_history)
    print("rules before validation", br.rules)
    _candidates = [(2,), (3,), (5,), (2,3), (2,5), (3,5)]
    threshold = 3/4
    print("candidates {} and minConfidence is {:.2f}"
          "".format(_candidates, threshold))
    _out = br.validation_rules((2,3,5), _candidates, threshold)
    print("rules after validation", br.rules)
    print("accepted rhs", _out)
```

Et voici ce que l'on peut observer

```
rules before validation []
candidates [(2,), (3,), (5,), (2, 3), (2, 5), (3, 5)] and minConfidence is
0.75
rules after validation [((3, 5), (2,)), ((2, 3), (5,))]
accepted rhs [(2,), (5,)]
```

build_rules

Notre version de cette méthode sera une forme itérative et non récursive de l'algorithme, d'une part pour des raisons d'efficacité et d'autre part, parce que `validation_rules` renvoie les parties droites acceptées (en fonction du seuil fixé)
La méthode reçoit en entrée un *itemset* de référence, la liste des *itemsets* de taille **1** qui le compose, l'indice de confiance minimal. Elle ne renvoie rien. Son objectif est de construire toutes les règles acceptables issues de l'*itemset* de référence.

Le code algorithmique devient donc

```
soit k la taille de l'itemset
sz_rhs = 1
tant que len(liste_rhs)>1 et k>sz_rhs+1 faire
        sz_rhs = sz_rhs + 1
    liste_rhs = cross_product()
    liste_rhs = validation_rules()
fait
```

generate_rules

Nous pouvons maintenant construire la méthode principale, elle reçoit en entrée le seuil minimum de confiance, et ne renvoie rien.
Chaque appel à cette méthode provoque l'appel de la méthode `reset`
Elle va parcourir `self.liste_itemsets` qui est organisée en liste d'*itemsets* de même taille

```
    print("{0} generate_rules {0}".format('*=*'))
    data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
    db = Apriori(data)
    br = Arules(db.main(.5), db.support_history)
    for k in (.75, .5, .25, .1):
        print("min confiance", k)
        print("rules before generation", br.rules)
        _out = br.generate_rules( k )
        print("rules after generate_rules", br.rules)
        print("return of generate_rules ?", _out)
        print("*"*17)
```

Et la sortie écran

```
*=* generate_rules *=*
min confiance 0.75
rules before generation []
rules after generate_rules [((1,), (3,)), ((5,), (2,)), ((2,), (5,))]
return of generate_rules ? None
*****************
min confiance 0.5
rules before generation [((1,), (3,)), ((5,), (2,)), ((2,), (5,))]
rules after generate_rules [((3,), (1,)), ((1,), (3,)), ((3,), (2,)), ((2,),
(3,)), ((5,), (2,)), ((2,), (5,)), ((5,), (3,)), ((3,), (5,)), ((5,), (2,
3)), ((3,), (2, 5)), ((2,), (3, 5))]
return of generate_rules ? None
*****************
min confiance 0.25
rules before generation [((3,), (1,)), ((1,), (3,)), ((3,), (2,)), ((2,),
(3,)), ((5,), (2,)), ((2,), (5,)), ((5,), (3,)), ((3,), (5,)), ((5,), (2,
3)), ((3,), (2, 5)), ((2,), (3, 5))]
rules after generate_rules [((3,), (1,)), ((1,), (3,)), ((3,), (2,)), ((2,),
(3,)), ((5,), (2,)), ((2,), (5,)), ((5,), (3,)), ((3,), (5,)), ((5,), (2,
3)), ((3,), (2, 5)), ((2,), (3, 5))]
return of generate_rules ? None
*****************
min confiance 0.1
rules before generation [((3,), (1,)), ((1,), (3,)), ((3,), (2,)), ((2,),
(3,)), ((5,), (2,)), ((2,), (5,)), ((5,), (3,)), ((3,), (5,)), ((5,), (2,
3)), ((3,), (2, 5)), ((2,), (3, 5))]
rules after generate_rules [((3,), (1,)), ((1,), (3,)), ((3,), (2,)), ((2,),
(3,)), ((5,), (2,)), ((2,), (5,)), ((5,), (3,)), ((3,), (5,)), ((5,), (2,
3)), ((3,), (2, 5)), ((2,), (3, 5))]
return of generate_rules ? None
*****************
```

`main`

Cette méthode prend en paramètre un seuil minimal de confiance et renvoie un `DataFrame` pandas de 9 colonnes :

1. lhs : la partie gauche de la règle
2. rhs: la partie droite de la règle
3. lhs_support: le support de lhs
4. rhs_support: le support de rhs
5. support: le support de la règle
6. confidence: l'indice de confiance de la règle
7. lift: l'indice d'amélioration de la règle
8. leverage: l'indice de levier de la règle
9. conviction: l'indice de conviction de la règle

Attention lorsque la confiance est à 1, on stockera dans la `DataFrame` la valeur `np.inf` pour la conviction

Voici un petit code exemple

```
    data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
    db = Apriori(data)
    br = Arules(db.main(.5), db.support_history)
    for k in range(1, 7):
        _ = br.main(1/k)
        print("min confidence 1/{}={:.3f}".format(k,1/k))
        print(_.head())
        print('*'*7)
```

Et ce qu'on obtient comme sortie écran:

```
min confidence 1/1=1.000
    lhs   rhs  lhs_support  rhs_support  support  confidence      lift
leverage  conviction
0  (1,)  (3,)         0.50         0.75     0.50         1.0  1.333333
0.1250         inf
1  (5,)  (2,)         0.75         0.75     0.75         1.0  1.333333
0.1875         inf
2  (2,)  (5,)         0.75         0.75     0.75         1.0  1.333333
0.1875         inf
*******
min confidence 1/2=0.500
    lhs   rhs  lhs_support  rhs_support  support  confidence      lift
leverage  conviction
0  (3,)  (1,)         0.75         0.50     0.50    0.666667  1.333333
0.1250        1.50
1  (1,)  (3,)         0.50         0.75     0.50    1.000000  1.333333
0.1250         inf
2  (3,)  (2,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
3  (2,)  (3,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
4  (5,)  (2,)         0.75         0.75     0.75    1.000000  1.333333
0.1875         inf
*******
min confidence 1/3=0.333
    lhs   rhs  lhs_support  rhs_support  support  confidence      lift
leverage  conviction
0  (3,)  (1,)         0.75         0.50     0.50    0.666667  1.333333
0.1250        1.50
1  (1,)  (3,)         0.50         0.75     0.50    1.000000  1.333333
0.1250         inf
2  (3,)  (2,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
```

```
3  (2,)  (3,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
4  (5,)  (2,)         0.75         0.75     0.75    1.000000  1.333333
0.1875         inf
*******
min confidence 1/4=0.250
    lhs   rhs  lhs_support  rhs_support  support  confidence      lift
leverage  conviction
0  (3,)  (1,)         0.75         0.50     0.50    0.666667  1.333333
0.1250        1.50
1  (1,)  (3,)         0.50         0.75     0.50    1.000000  1.333333
0.1250         inf
2  (3,)  (2,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
3  (2,)  (3,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
4  (5,)  (2,)         0.75         0.75     0.75    1.000000  1.333333
0.1875         inf
*******
min confidence 1/5=0.200
    lhs   rhs  lhs_support  rhs_support  support  confidence      lift
leverage  conviction
0  (3,)  (1,)         0.75         0.50     0.50    0.666667  1.333333
0.1250        1.50
1  (1,)  (3,)         0.50         0.75     0.50    1.000000  1.333333
0.1250         inf
2  (3,)  (2,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
3  (2,)  (3,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
4  (5,)  (2,)         0.75         0.75     0.75    1.000000  1.333333
0.1875         inf
*******
min confidence 1/6=0.167
    lhs   rhs  lhs_support  rhs_support  support  confidence      lift
leverage  conviction
0  (3,)  (1,)         0.75         0.50     0.50    0.666667  1.333333
0.1250        1.50
1  (1,)  (3,)         0.50         0.75     0.50    1.000000  1.333333
0.1250         inf
2  (3,)  (2,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
3  (2,)  (3,)         0.75         0.75     0.50    0.666667  0.888889
-0.0625        0.75
4  (5,)  (2,)         0.75         0.75     0.75    1.000000  1.333333
0.1875         inf
*******
```

L'intérêt de cette forme et que nous pourrons utiliser la bibliothèque `pandas` dans la seconde partie du TP pour pouvoir extraire les règles les plus pertinentes.