# TP01 les itemsets fréquents

## La classe `Apriori` (fichier apriori.py)

(Re)Lisez la description informelle faites en cours sur le fonctionnement de cet algorithme, puis lisez toute la fiche afin de bien comprendre le découpage avant de vous lancer dans le codage à proprement parler. Pensez, après chaque méthode, à vérifier sur un petit exemple, le bon fonctionnement de votre implémentation et ne commencez pas la méthode suivante tant que cela ne fonctionne pas correctement. N'oubliez pas de mettre la signature et n'hésitez pas à mettre un docstring (cela vous aidera à mieux vous rappelez à quoi servent telle ou telle méthode dans quelque temps).

L'objectif de cette classe est, étant donné une série de transactions, de calculer (une estimation) des probabilités d'occurrences des items dans une transaction. On commence par les occurrences simples (un item), puis de 2 items, etc jusqu'à $n$, le calcul s'arrête lorsque l'on a vu toutes les combinaisons dépassant un seuil minimal, fixé a priori.

### constructeur

Le constructeur reçoit un dictionnaire python, dont les clefs correspondent au `tid` et dont les valeurs sont des listes d'entiers correspondant aux articles (*items*).
Le dictionnaire est stocké dans `self.dbase` puis le constructeur appelle la méthode `reset`

### reset

méthode sans paramètre qui ne renvoie rien, elle va initialiser différentes variables :

- `self.candidates_sz` à 1, permet de connaître la taille des *itemsets* à traiter
- `self.current` un dictionnaire, dont les clefs sont des *itemsets* et dont les valeurs sont des ensembles de `tid`. À l'initialisation pour chaque tuple de taille 1 on met un `set` de la liste des transactions où l'item apparaît
- `self.candidates` un dictionnaire dont les clefs sont les `tid` et les valeurs des listes d'*itemsets* de taille 1
- `self.support_history` un dictionnaire vide

Le code suivant:

```
data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
db = Apriori(data)
print("dbase", db.dbase)
print("candidates_sz", db.candidates_sz)
print("support_history", db.support_history)
```

```
print("candidates", db.candidates)
print("current", db.current)
```

doit produire quelque chose comme:

```
dbase {100: [1, 3, 4], 200: [2, 3, 5], 300: [1, 2, 3, 5], 400: [2, 5]}
candidates_sz 1
support_history {}
candidates {100: [(1,), (3,), (4,)], 200: [(2,), (3,), (5,)], 300: [(1,),
(2,), (3,), (5,)], 400: [(2,), (5,)]}
current {(1,): {100, 300}, (3,): {200, 100, 300}, (4,): {100}, (2,): {200,
300, 400}, (5,): {200, 300, 400}}
```

## support

méthode qui reçoit en entrée un réel entre 0 et 1, renvoie un dictionnaire construit à partir du dictionnaire `self.current` dont les clefs sont des ***itemsets*** et dont les valeurs sont les supports (voir support de cours). Il s'agit donc d'un filtre qui va éliminer les co-occurrences trop rares.

```
data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
db = Apriori(data)
print("support(.7)", db.support(.7))
print("support(.2)", db.support(.2))
```

devrait produire :

```
support(.7) {(3,): 0.75, (2,): 0.75, (5,): 0.75}
support(.2) {(1,): 0.5, (3,): 0.75, (4,): 0.25, (2,): 0.75, (5,): 0.75}
```

## scan_dbase

méthode qui reçoit en entrée un réel entre 0 et 1, cette méthode ne renvoie rien, par contre elle va :

1. mettre à jour le dictionnaire `self.support_history` en intégrant le dictionnaire renvoyé par la méthode `self.support(minsupp)` . La mise à jour se fera par la méthode `update` des dictionnaires
2. modifie `self.current` en ne gardant que les informations pertinentes

Une fois calculé le support, on enlève de la variable `self.current` les co-occurrences qui n'ont pas un support suffisant.

```
>>> d = {'a':2, 'b': 4, 'c':5}
>>> e = {'a':13, 'd':47}
>>> d.update(e)
```

```
>>> d
{'a': 13, 'b': 4, 'c': 5, 'd': 47}
>>> e
{'a': 13, 'd': 47}
```

Ainsi après le code suivant :

```
print("#{}#".format('='*73))
db.scan_dbase(.7)
print("support_history", db.support_history)
print("current", db.current)
```

On doit obtenir les résultats suivants :

```
#=========================================================================#
support_history {(3,): 0.75, (2,): 0.75, (5,): 0.75}
current {(3,): {200, 100, 300}, (2,): {200, 300, 400}, (5,): {200, 300,
400}}
```

## Lk

méthode qui ne reçoit rien mais qui renvoie la liste triée des clefs de `self.current`

```
print("Lk for size {}".format(db.candidates_sz), db.Lk())

Lk for size 1 [(2,), (3,), (5,)]
```

## cross_product

méthode sans paramètre qui ne renvoie rien, mais qui modifie plusieurs variables. C'est le **gros** morceau de l'algorithme dont voici une description rapide

```
récupérer les itemsets de l'itération précédente (Lk)
récupérer la taille des itemsets (k)
récupérer le nombre d'itemsets (p)
creer futur = {}
pour i de 1 à p-1
   j = i+1
   tant que (j <= p
         ET
         les k-1 premières valeurs de itemset[i] et itemset[j] sont
identiques)
      construire nouveau = itemset[i] + itemset[j][-1]
      si tous les sous-items de nouveau de longueur k sont dans Lk alors
         futur[nouveau] = intersection(tid de itemset[i], tid de itemset[j])
      fsi
      j = j+1
```

```
    ftq
current <- futur
mettre à jour candidates
mettre à jour candidates_sz
```

**Danger**

L'algorithme compte à partir de 1, python compte à partir de 0
Un *itemset* est un tuple, on ne peut pas ajouter un élément dans un tuple
En python l'intersection de 2 ensembles se fait comme ceci

```
>>> x
{1, 2, 3}
>>> y
{1, 3, 5}
>>> z = x.intersection(y)
>>> z
{1, 3}
```

Et voici ce qui se passe après un appel à cross_product

```
data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
db = Apriori(data)
print("dbase", db.dbase)
print("candidates_sz", db.candidates_sz)
print("support_history", db.support_history)
print("candidates", db.candidates)
print("current", db.current)
print("support(.7)", db.support(.7))
print("support(.2)", db.support(.2))
print("#{}#".format('='*73))
db.scan_dbase(.7)
print("support_history", db.support_history)
print("current", db.current)
print("Lk for size {}".format(db.candidates_sz), db.Lk())
print("#{}#".format('='*73))
db.cross_product()
print("dbase", db.dbase)
print("candidates_sz", db.candidates_sz)
print("support_history", db.support_history)
print("candidates", db.candidates)
print("current", db.current)
```

On obtient :

```
dbase {100: [1, 3, 4], 200: [2, 3, 5], 300: [1, 2, 3, 5], 400: [2, 5]}
candidates_sz 1
support_history {}
candidates {100: [(1,), (3,), (4,)], 200: [(2,), (3,), (5,)], 300: [(1,),
(2,), (3,), (5,)], 400: [(2,), (5,)]}
current {(1,): {100, 300}, (3,): {200, 100, 300}, (4,): {100}, (2,): {200,
300, 400}, (5,): {200, 300, 400}}
support(.7) {(3,): 0.75, (2,): 0.75, (5,): 0.75}
support(.2) {(1,): 0.5, (3,): 0.75, (4,): 0.25, (2,): 0.75, (5,): 0.75}
#=========================================================================#
support_history {(3,): 0.75, (2,): 0.75, (5,): 0.75}
current {(3,): {200, 100, 300}, (2,): {200, 300, 400}, (5,): {200, 300,
400}}
Lk for size 1 [(2,), (3,), (5,)]
#=========================================================================#
dbase {100: [1, 3, 4], 200: [2, 3, 5], 300: [1, 2, 3, 5], 400: [2, 5]}
candidates_sz 2
support_history {(3,): 0.75, (2,): 0.75, (5,): 0.75}
candidates {200: [(2, 3), (2, 5), (3, 5)], 300: [(2, 3), (2, 5), (3, 5)],
400: [(2, 5)]}
current {(2, 3): {200, 300}, (2, 5): {200, 300, 400}, (3, 5): {200, 300}}
```

## main

La méthode main, prend en paramètre un réel entre 0 et 1 (la valeur du support minimum), et produit en sortie une liste de listes d'*itemsets*

1. elle appelle la méthode `self.reset()` histoire d'éviter les mauvaises manips
2. elle itère `scan_db` stockage de `Lk` et `cross_product`

```
data = {100:[1, 3, 4], 200:[2, 3, 5], 300:[1, 2, 3, 5], 400:[2, 5]}
db = Apriori(data)
print("dbase", db.dbase)
print("candidates_sz", db.candidates_sz)
print("support_history", db.support_history)
print("candidates", db.candidates)
print("current", db.current)
print("main(.3)", db.main(.3))
print("candidates_sz", db.candidates_sz)
print("support_history", db.support_history)
print("candidates", db.candidates)
print("current", db.current)
```

Dont la sortie est :

```
dbase {100: [1, 3, 4], 200: [2, 3, 5], 300: [1, 2, 3, 5], 400: [2, 5]}
candidates_sz 1
support_history {}
```

```
candidates {100: [(1,), (3,), (4,)], 200: [(2,), (3,), (5,)], 300: [(1,),
(2,), (3,), (5,)], 400: [(2,), (5,)]}
current {(1,): {100, 300}, (3,): {200, 100, 300}, (4,): {100}, (2,): {200,
300, 400}, (5,): {200, 300, 400}}
main(.3) [[(1,), (2,), (3,), (5,)], [(1, 3), (2, 3), (2, 5), (3, 5)], [(2,
3, 5)]]
candidates_sz 3
support_history {(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}
candidates {200: [(2, 3, 5)], 300: [(2, 3, 5)]}
current {(2, 3, 5): {200, 300}}
```

seconde partie: les règles d'association