td2a_cenonce_session_5_donnees_non_structurees_et_programmation_fon

June 24, 2022

1 2A.i - Données non structurées et programmation fonctionnelle

Calculs de moyennes et autres statistiques sur une base twitter au format JSON avec de la programmation fonctionnelle (module cytoolz).

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

Commencez par télécharger la base de donnée twitter_for_network_100000.db. Vous pourrez éventuellement télécharger la base complète (3,4 millions d'utilisateurs, plutôt que 100000) ultérieurement si vous souhaitez tester vos fonctions. Ne perdez pas de temps avec ceci dans ce TP : twitter_for_network_full.db. Vous pouvez consulter l'aide de pytoolz (même interface que cytoolz). La section sur l'API est particulièrement utile car elle résume bien les différentes fonctions.
Liens alternatifs :

- twitter_for_network_100000.db.zip
- twitter_for_network_full.db.zip

Ensuite exécutez la cellule suivante :

[2]:
```
import pyensae.datasource
pyensae.datasource.download_data("twitter_for_network_100000.db.zip")
```

[2]:
```
['twitter_for_network_100000.db']
```

[3]:
```
import cytoolz as ct
import cytoolz.curried as ctc
import sqlite3
import pprint
import json

conn_sqlite = sqlite3.connect("twitter_for_network_100000.db")
cursor_sqlite = conn_sqlite.cursor()
```

1.1 Description de la base de donnée

Nous nous intéresserons à 3 tables : **tw_users**, **tw_status** et **tw_followers_id**.
La première (**tw_users**) contient des profils utilisateurs tels que retournés par l'api twitter (à noter que les profils ont été "épurés" d'informations jugées inutiles pour limiter la taille de la base de donnée).
La deuxième (**tw_status**) contient des status twitter (tweet, retweet, ou réponse à un tweet), complets, issus d'une certaine catégorie d'utilisateurs (les tweets sont tous issus d'environ 70 profils).
La troisième (**tw_followers_id**) contient des listes d'id d'users, qui suivent les utilisateurs référencés par la colonne user_id. Là encore ce ne sont les followers que de environ 70 profils. Chaque entrée contient au plus 5000 id de followers (il s'agit d'une limitation de twitter).
Elles ont les structures suivantes :

```
CREATE TABLE tw_followers_id( user_id bigint NOT NULL, cursor bigint NOT NULL,
                              prev_cursor bigint NOT NULL, next_cursor bigint NOT NULL,
                              update_time timestamp NOT NULL, content json NOT NULL);
CREATE TABLE tw_users( id bigint NOT NULL, last_update timestamp NOT NULL,
                       content json, screen_name character varying(512));
CREATE TABLE tw_status( id bigint NOT NULL, user_id bigint NOT NULL,
                        last_update timestamp NOT NULL, content json);
```

Les trois possèdent un champ content, de type json, qui sera celui qui nous interessera le plus. Vous pouvez accédez aux données dans les tables avec les syntaxes suivantes (vous pouvez commenter/décommenter les différentes requêtes).

[4]:
```
# cursor_sqlite.execute("SELECT user_id, content FROM tw_followers_id")
cursor_sqlite.execute("SELECT id, content, screen_name FROM tw_users")
# cursor_sqlite.execute("SELECT id, content, user_id FROM tw_status")

for it_elt in cursor_sqlite:
    ## do something here
    pass

# ou, pour accéder à un élément :
cursor_sqlite.execute("SELECT id, content, screen_name FROM tw_users")
cursor_sqlite.fetchone()
```

[4]:
```
(1103159180,
 '{"utc_offset": 7200, "friends_count": 454, "entities": {"description":
{"urls": []}, "url": {"urls": [{"expanded_url": "http://www.havas.com",
"display_url": "havas.com", "indices": [0, 22], "url":
"http://t.co/8GcZtydjWh"}]}}, "description": "Havas Group CEO", "id":
1103159180, "contributors_enabled": false, "geo_enabled": false, "name":
"Yannick Bollor\\u00e9", "favourites_count": 873, "verified": true, "protected":
false, "created_at": "Sat Jan 19 08:23:33 +0000 2013", "statuses_count": 654,
"lang": "en", "time_zone": "Ljubljana", "screen_name": "YannickBollore",
"location": "", "id_str": "1103159180", "url": "http://t.co/8GcZtydjWh",
"followers_count": 7345, "listed_count": 118, "has_extended_profile": false}',
 'YannickBollore')
```

Toutefois les curseurs de base de donnée en python se comportent comme des "iterables" (i.e. comme une liste ou une séquence, mais sans nécessairement charger toutes les données en mémoire). On peut donc les passer directement en argument aux fonctions de cytoolz.

[5]:
```
cursor_sqlite.execute( "SELECT user_id, content FROM tw_followers_id")
print( ct.count( cursor_sqlite ) )
cursor_sqlite.execute( "SELECT id, content, user_id FROM tw_status")
print( ct.count( cursor_sqlite ) )
```

```
cursor_sqlite.execute( "SELECT id, content, screen_name FROM tw_users")
print( ct.count( cursor_sqlite ) )
```

```
2079
16092
100071
```

Attention au fait que le curseur garde un état.
Par exemple exécutez le code suivant :

[6]:
```
cursor_sqlite.execute( "SELECT user_id, content FROM tw_followers_id")
print( ct.count( cursor_sqlite ) )
print( ct.count( cursor_sqlite ) )
```

```
2079
0
```

Le deuxième count renvoit 0 car le curseur se rappelle qu'il est déjà arrivé à la fin des données qu'il devait parcourir. Il faut donc réinitialiser le curseur :

[7]:
```
cursor_sqlite.execute( "SELECT user_id, content FROM tw_followers_id")
print( ct.count( cursor_sqlite ) )
cursor_sqlite.execute( "SELECT user_id, content FROM tw_followers_id")
print( ct.count( cursor_sqlite ) )
```

```
2079
2079
```

On peut également mettre la commande execute à l'intérieur d'une fonction, que l'on appelle ensuite :

[8]:
```
def get_tw_followers_id():
    return cursor_sqlite.execute( "SELECT user_id, content FROM tw_followers_id")
print( ct.count( get_tw_followers_id() ) )
print( ct.count( get_tw_followers_id() ) )
```

```
2079
2079
```

La commande exécute en elle-même ne prend pas du tout de temps, car elle ne fait que préparer la requête, n'hésitez donc pas à en mettre systématiquement dans vos cellules, plutôt que de risquer d'avoir un curseur dont vous ne vous souvenez plus de l'état.

1.2 Partie 1 - description de la base de donnée

Question 1 - éléments unique d'une table Trouvez la liste des `user_id` différents dans la table `tw_followers_id`, en utilisant les fonctions cytoolz.
La fonction qui pourra vous être utiles ici :

- ct.unique(seq) ⇒ à partir d'une séquence, renvoit une séquence où tous les doublons ont été supprimés

Vous vous rappelez sans doute que nous utilisions systématiquement pluck et map pour les exemples du cours, ceux-ci ne sont pas nécessaires ici. A noter qu'il faudra sans doute utilisez la fonction `list( ... )`, ou une boucle `for` pour forcer l'évaluation des fonctions cytoolz.

[9]:
```
import cytoolz as ct
import cytoolz.curried as ctc
```

A noter que si vous voyez apparaître vos résultats sous la forme (79145543,), c'est normal, le curseur sqlite renvoit toujours ces résultats sous forme de tuple : *(colonne1, colonne2, colonne3, …)* et ce même si il n'y a qu'une seule colonne dans la requête. Nous utiliserons pluck pour extraire le premier élément du tuple.

Question 2 - nombre d'élements unique d'une table Trouvez le nombre de user_id différents dans la table tw_followers_id, en utilisant les fonctions cytoolz.
Les fonctions qui pourront vous êtres utiles ici :

- ct.count(seq) => compte le nombre d'éléments d'une séquence
- ct.unique(seq) => à partir d'une séquence, renvoit une séquence où tous les doublons ont été supprimés

Vous vous rappelez sans doute que nous utilisions systématiquement pluck et map pour les exemples du cours, ceux-ci ne sont pas nécessaires, ici.

[10]:
```
import cytoolz as ct
import cytoolz.curried as ctc
```

Question 3 : création d'une fonction comptez_unique A l'aide de ct.compose, créez une fonction `comptez_unique` qui effectue directement cette opération. Pour rappel, `ct.compose( f, g, h, ...)` renvoit une fonction qui appelée sur `x` exécute `(f(g(h(x)))`. ct.compose prend un nombre d'arguments quelconque. A noter que les fonctions données en argument doivent ne prendre qu'un seul argument, ce qui est le cas ici. Pensez bien que comme vous manipulez ici les fonctions elle-même, il ne faut pas mettre de parenthèses après

[11]:
```
import cytoolz as ct
import cytoolz.curried as ctc
```

[12]:
```
## Pour tester votre code, cette ligne doit renvoyer le même nombre qu'à la question 2
# comptez_unique( cursor_sqlite.execute( "SELECT user_id FROM tw_followers_id") )
```

Question 4 : compte du nombre de valeurs de "location" différentes dans la table tw_users
Nous allons utiliser la fonction comptez_unique définie précédemment pour comptez le nombre de "location" différentes dans la table **tw_users**.
Pour cela il faudra faire appel à deux fonctions :

- ct.pluck pour extraire une valeur de tous les éléments d'une séquence
- **ct.map** (ie `map = cytoolz.curry(map)`) pour appliquer une fonction (ici json.loads pour transformer une chaîne de caractère au format json en objet python).

Il faudra sans doute appliquer ct.pluck deux fois, une fois pour extraire la colonne content du résultat de la requête (même si celle-ci ne comprend qu'une colonne) et une fois pour extraire le champ "location" du json.
Les syntaxes de ces fonctions sont les suivantes :

- `ct.pluck( 0, seq )` (cas d'une séquence de liste ou de tuple) ou `ct.pluck( key, seq )` (cas d'une séquence de dictionnaire).
- `ct.map( f, seq )` où f est la fonction que l'on souhaite appliquer (ne mettez pas les parenthèses après le f, ici vous faites références à la fonction, pas son résultat)

Astuce : dans le cas improbable où vous auriez un ordinateur sensiblement plus lent que le rédacteur du tp, rajoutez LIMIT 10000 à la fin des requêtes

[13]:
```
import cytoolz as ct

cursor_sqlite.execute( "SELECT content FROM tw_users")
# Le résultat attendu est 13730
```

[13]:
```
<sqlite3.Cursor at 0x25eaa7aae30>
```

Question 5 : curly fonctions Comme on risque de beaucoup utiliser les fonctions `ct.map` et ct.pluck, on veut se simplifier la vie en utilisant la notation suivante :

[14]:
```
pluck_loc = ctc.pluck("location")
map_loads = ctc.map(json.loads)
pluck_0   = ctc.pluck(0)
```

```
c:\python35_x64\lib\site-packages\ipykernel\__main__.py:1: DeprecationWarning:
inspect.getargspec() is deprecated, use inspect.signature() instead
  if __name__ == '__main__':
c:\python35_x64\lib\site-packages\ipykernel\__main__.py:3: DeprecationWarning:
inspect.getargspec() is deprecated, use inspect.signature() instead
  app.launch_new_instance()
```

Notez bien que nous utilisons ctc.pluck et non pas ct.pluck, car le package cytoolz.curry (ici importé en temps que ctc) contient les versions de ces fonctions qui supportent l'évaluation partielle.
Les objets **pluck_loc**, **map_loads**, **pluck_0** sont donc des fonctions à un argument, construites à partir de fonctions à deux arguments. Utilisez ces 3 fonctions pour simplifier l'écriture de la question 4

[15]:
```
import cytoolz as ct

cursor_sqlite.execute( "SELECT content FROM tw_users")
# Le résultat attendu est 13730
```

[15]:
```
<sqlite3.Cursor at 0x25eaa7aae30>
```

Question 6 : fonction get_json_seq A partir des fonctions précédentes et de la fonction compose, créez une fonction get_json_seq, qui à partir d'un curseur d'une requête dont la colonne content est en première position, renvoit une séquence des objets json loadés.
Vous devez pouvoir l'utiliser pour réécrire le code de la question précédente ainsi :

[16]:
```
import cytoolz as ct

cursor_sqlite.execute( "SELECT content FROM tw_users")
# comptez_unique( pluck_loc( get_json_seq(cursor_sqlite)))
```

[16]:
```
<sqlite3.Cursor at 0x25eaa7aae30>
```

Question 7 : liste des localisations avec Paris On peut vérifier si une localisation contient le mot "Paris", avec toutes ces variations de casse possible avec la fonction suivante :

[17]:
```
def contains_paris(loc):
    return "paris" in loc.lower()
```

En utilisant cette fonction et la fonction `ct.filter`, trouvez :

- le nombre d'utilisateur dont la location contient Paris sous une forme ou une autre (question 7.1)
- tous les variantes de location contenant Paris (pour info il y en a 977)

ct.filter s'utilise avec la syntaxe `ct.filter( f, seq )` (voir filter) et renvoit une séquence de tous les éléments de la séquence en entrée pour lesquels f renvoit true. Vous aurez besoin des fonctions ct.unique et ct.count. Si vous avez une sortie du type `<cytoolz.itertoolz._unique_identity at 0x7f3e7f3d6d30>`, rajouter la fonction `list( ... )` autour pour forcer l'évaluation.

[18]:
```
## Question 7.1
import cytoolz as ct

cursor_sqlite.execute( "SELECT content FROM tw_users")
## le résultat attendu est 5470
```

[18]:
```
<sqlite3.Cursor at 0x25eaa7aae30>
```

[19]:
```
## Question 7.2
import cytoolz as ct

cursor_sqlite.execute( "SELECT content FROM tw_users")
## la liste doit contenir 977 éléments
```

[19]:
```
<sqlite3.Cursor at 0x25eaa7aae30>
```

Question 8 : somme des tweets de tous les utilisateurs dont la location contient Paris Calculez le nombre de tweets total par les utilisateurs dont la *location* contient Paris.
Dans le json de twitter, la clé pour cela est `statuses_count`. Pour cela plusieurs possibilités :

- la plus simple est de redéfinir une fonction `contains_paris`, qui prenne en entrée un user json
- groupby *("location", seq)* vous renvoit les réponses groupées par location. Cette méthode possède l'inconvénient de charger toutes les données en mémoire
- reduceby *("location", lambda x,y: x + y["statuses_count"], seq, 0)* vous renvoit la somme par location, il ne reste plus qu'à filtrer et additionner
- pluck *(["location", "statuses_count"], seq)* vous permet de garder les deux informations. Il faudra changer la fonction contains paris pour celle suivante (`contains_paris_tuple`)

Réponse attendue : 9811612

[20]:
```
import cytoolz as ct
```

Question 9 : comparaison des followers d'homme politique On va maintenant s'intéresser à la proximité / corrélation entre les hommes politiques, que l'on mesurera à partir de la formule :
$\frac{1}{2} * (\frac{nbFollowersCommun}{nbFollowersHommePolitique_1} + \frac{nbFollowersCommun}{nbFollowersHommePolitique_2})$
On prend donc la moyenne des ratios des followers de chaque homme politique suivant l'autre (cette formule semble s'accommoder assez bien des différences du nombre de followers entre homme politiques). On s'intéressera notamment aux hommes politiques suivants :

```
benoithamon | 14389177
montebourg  | 69255422
alainjuppe  | 258345629
```

De fait vous pouvez prendre n'importe quel homme ou femme politique, les résultats de cette méthode sont assez probants malgré sa rusticité.
Important : pensez à appliquer la cellule ci-dessous

[21]:
```
try:
    cursor_sqlite.execute("CREATE UNIQUE INDEX tw_users_id_index ON tw_users(id)")
    print("Index created")
except sqlite3.OperationalError as e:
    if( "index tw_users_id_index already exists" in str(e)):
        print("Ok, index already exists")
    else:
        raise e
```

```
Index created
```

La façon la plus simple est de charger les listes d'id de followers en mémoire, dans des objets de type set, et de les comparer avec les opérateurs & (intersection) - (différences). On peut aussi chercher une méthode approchée, en comparant de façon aléatoire les listes contenues dans `tw_follower_id`.
Tips : si vous trouvez que Montebourg est plus proche de Juppé que de Hamon, vous vous êtes planté …

[22]:
```

```

1.3 Partie 2 : avec dask

Essayez d'exécuter le code suivant

[23]:
```
import dask
```

dask peut vous permettre de paralléliser de façon efficace votre code entre plusieurs processeurs. Utilisez le code suivant pour splitter la base *twitter_for_network_full.db* en plusieurs fichiers plats (NB: pensez à nettoyer votre disque dur après ce tp).

[24]:
```
import cytoolz as ct # import groupby, valmap, compose
import cytoolz.curried as ctc ## pipe, map, filter, get
import sqlite3
import pprint
try:
    import ujson as json
except:
    import json

conn_sqlite_f = sqlite3.connect("twitter_for_network_100000.db")
cursor_sqlite_f = conn_sqlite_f.cursor()
```

[25]:
```
cursor_sqlite_f.execute("SELECT content FROM tw_users")

for it in range(100):
    with open( "tw_users_split_{0:02d}.json".format(it), 'w') as f:
        for it_index, it_json in enumerate( cursor_sqlite_f ):
            f.write( it_json[0] )
            f.write("\n")
            if it_index == 100000:
                break
        else:
            break
```

Calculez maintenant, en utilisant dask.bag :

- le nombre total de status

- le nombre de status moyen par location
- la distribution du nombre de followers par puissance de 10 sur l'ensemble des users

[26]:
```
## Code commun nécessaire

import dask.bag as dbag
try:
    import ujson as json
except:
    print("ujson unavailable")
    import json
from operator import add

a = dbag.read_text('tw_users_split*.json')
```

```
ujson unavailable
```

[27]:
```
# Le nombre total de status
```

[28]:
```
# Le nombre moyen de tweet par location.
import cytoolz
import cytoolz.curried as ctc
```

[29]:
```
# La distribution du nombre de followers par puissance de 10
import math
```

[30]:
```

```