Master Informatique, parcours MALIA

Carnets de note Python pour le cours de Network Analysis for Information Retrieval

Julien Velcin, laboratoire ERIC, Université Lyon 2

Représentation des documents (partie 1)

Nous allons voir :

- Considérations générales
- Modèle du sac de mots (*bag of words*) et matrice Documents x Termes
- Extraire des caractéristiques (tokenisation, prétraitements)
- Schémas de pondération (TF, TFxIDF...)
- Comparer deux textes
- Application : construire son propre moteur de recherche

Considérations générales

Il existe differentes manières de représenter des données textuelles :

- Chaîne de caractères (string)
- Bag-of-Words (BoW)
- Vector Space Model (VSM)
- Séquence de mots
- Ajouter des méta-données (par ex. catégories grammaticales)
- Représentations plus complexes : arbres syntaxiques, graphes, etc.

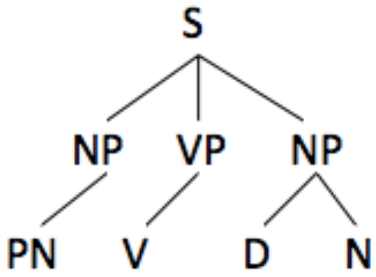
Chaque représentation implique une manière différente de *comparer* les documents.

Un exemple très simple

"*John Doe has bought an apple.*"

Point de vue linguistique:

"John Doe has bought an apple."



Point de vue statistique :

"John Doe has bought an apple."

{ apple,

bought,

John_Doe }

I love holidays. Sunbathing, swimming… I cannot imagine being away from the sea during holidays. Going to the mountain is not the same. I do not know…. I think the mountain is better for winter holidays and the sea for the summer ones.

word	*Frequency*
I	4
love	1
holidays	3
…	
sea	2
for	2
the	6
summer	1
ones	1

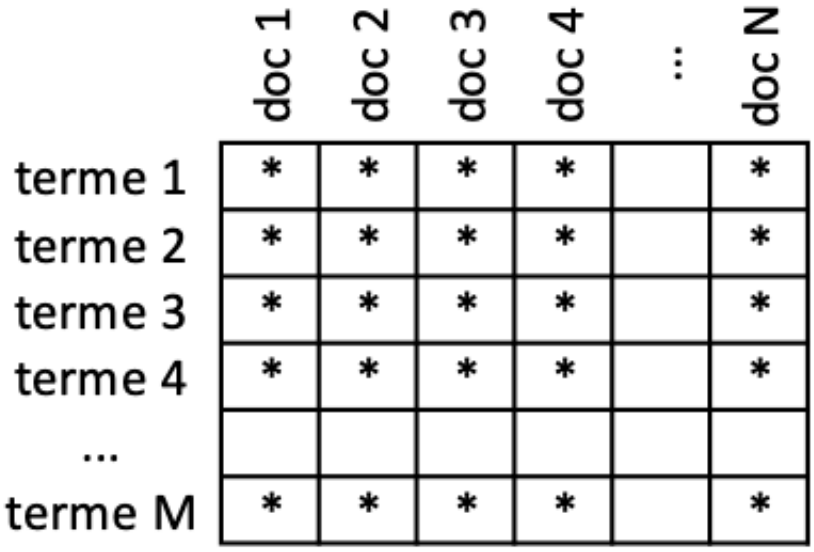
On peut déjà observer des problèmes évidents, tel que :

"Mary asked Fred out."
"Fred asked Mary out."

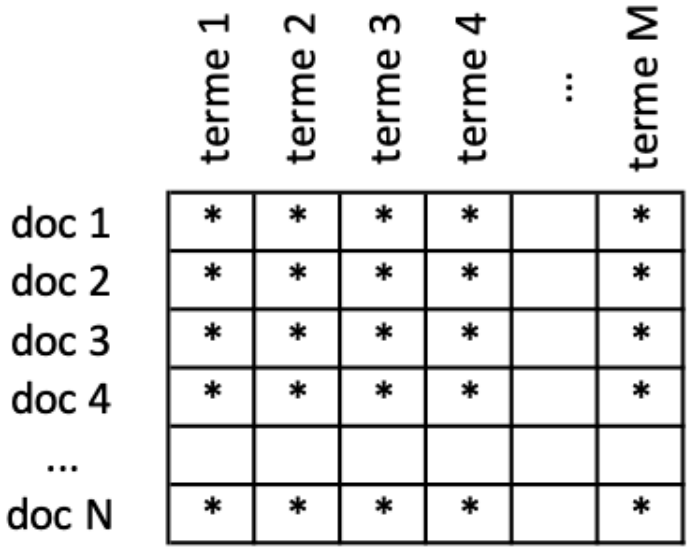
La représentation est la même :

word	***Frequency***
Mary	1
asked	1
Fred	1
out	1

A partir d'un *ensemble* de documents, on souhaite obtenir l'objet suivant :



ou sa transposée : la



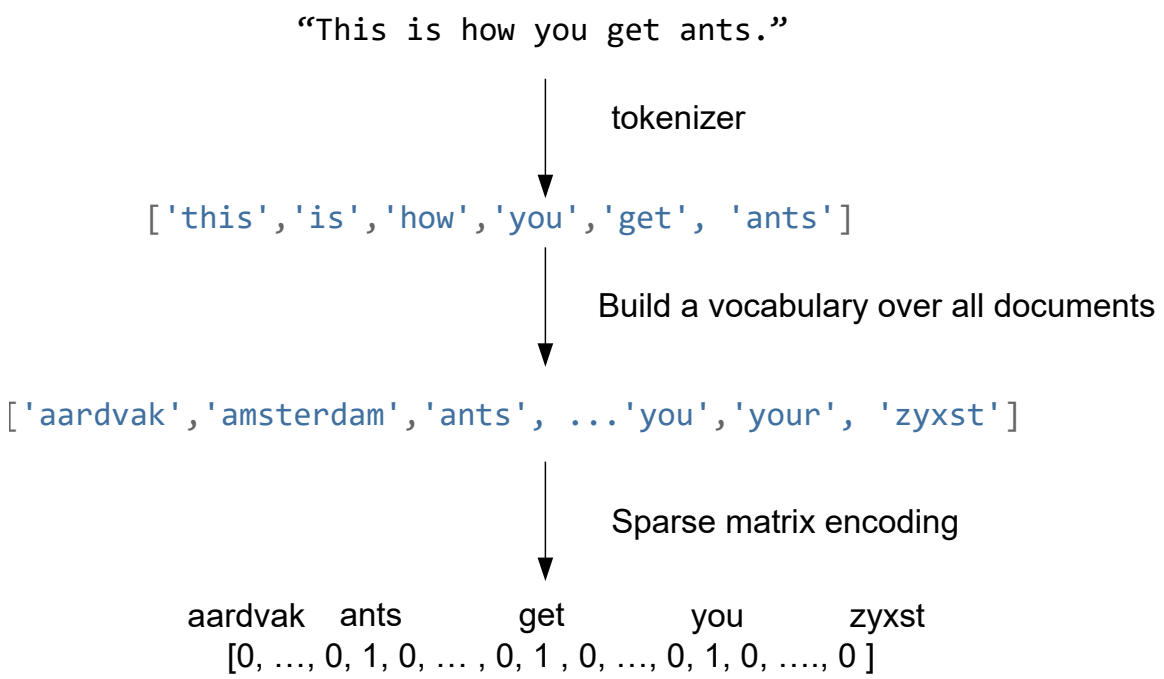
matrice Documents x Termes :

	D1	D2	D3	D4	D5	D6	D7	D8	D9
measur	1	0	0	2	1	0	1	0	0
effici	1	0	1	0	0	0	1	0	0
machin	1	0	0	0	0	0	0	0	0
factori	1	0	0	0	0	0	0	0	0
system	1	0	0	0	0	0	0	0	0
input	1	0	0	2	0	0	1	0	1
output	1	1	0	2	0	0	1	0	1
averag	0	1	0	0	0	0	1	0	0
cost	0	1	1	0	0	0	0	0	0
resourc	0	1	0	0	0	0	0	0	0
consum	0	1	0	0	0	0	0	1	0
econom	0	0	0	1	0	0	0	0	0
labor	0	0	0	1	0	0	0	0	0
revenu	0	0	0	1	0	0	0	0	0
gdp	0	0	0	1	1	0	0	0	0
predict	0	0	0	0	1	0	0	0	0
futur	0	0	0	0	1	0	0	0	0
growth	0	0	0	0	1	0	0	0	0
gain	0	0	0	0	0	1	0	0	0
accomplish	0	0	0	0	0	1	0	0	0
energi	0	0	0	0	0	0	0	1	0
produc	0	0	0	0	0	0	0	1	0
food	0	0	0	0	0	0	0	1	0

Par exemple :

Extraire les caractéristiques des textes avec le modèle "sac de mots"

En pratique, on utilise le formalisme du "sac de mots" et on suit plusieurs étapes :



Prenons à présent deux documents, tels que :

In [2]:
```
X = ["Some say the world will end in fire,",
     "Some say in ice."]
```

In [3]:
```
len(X)
```

Out[3]: 2

La librairie contient une fonction qui permet de "vectoriser" un ensemble de textes en prenant en compte un certain nombre de prétraitements (*preprocessing*) couramment employés : mis en minuscule, utilisation de mots-outils, etc.

In [4]:
```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectorizer.fit(X)
```

Out[4]: ▾ CountVectorizer

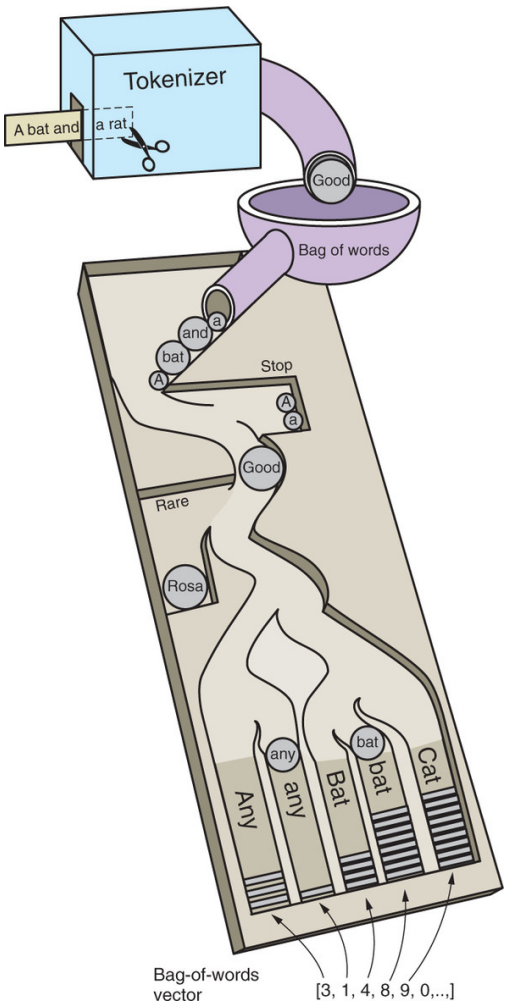
CountVectorizer

Observons le vocabulaire automatiquement construit à partir de ces deux textes :

In [5]:
```
vectorizer.vocabulary_
```

Out[5]:
```
{'some': 5,
 'say': 4,
 'the': 6,
 'world': 8,
 'will': 7,
 'end': 0,
 'in': 3,
 'fire': 1,
 'ice': 2}
```

Une illustration bien utile, tirée de : Natural Language Processing in Action: Understanding, analyzing, and generating text with Python, par Hobson Lane, Cole Howard, Hannes Hapke, 2019, ISBN 9781617294631 (https://www.manning.com/books/natural-language-processing-in-action) :



Matrice documents x termes

On peut maintenant construire la matrice documents * termes :

In [6]:
```
X_bag_of_words = vectorizer.transform(X)
```

In [7]:
```
X_bag_of_words
```

Out[7]:
```
<2x9 sparse matrix of type '<class 'numpy.int64'>'
        with 12 stored elements in Compressed Sparse Row format>
```

In [8]:
```
X_bag_of_words.shape
```

Out[8]:
```
(2, 9)
```

Jetons un oeil au contenu de la matrice :

In [9]:
```
X_bag_of_words.toarray()
```

Out[9]:
```
array([[1, 1, 0, 1, 1, 1, 1, 1, 1],
       [0, 0, 1, 1, 1, 1, 0, 0, 0]])
```

On observe que la valeur indiquée dans une cellule est le nombre d'occurrences d'un terme dans un document (TF pour *Term Frequency*)

On peut également retrouver le nom des termes du vocabulaire :

In [10]:
```
vectorizer.get_feature_names_out()
```

Out[10]:
```
array(['end', 'fire', 'ice', 'in', 'say', 'some', 'the', 'will', 'world'],
      dtype=object)
```

On peut retrouver les attributs (features) utilisés dans les documents :

In [11]:
```
vectorizer.inverse_transform(X_bag_of_words)
```

Out[11]:
```
[array(['end', 'fire', 'in', 'say', 'some', 'the', 'will', 'world'],
       dtype='<U5'),
 array(['ice', 'in', 'say', 'some'], dtype='<U5')]
```

Représentation TFxIDF

A la place du nombre d'occurrences (TF), on peut utiliser une autre mesure qui prend en compte la rareté d'un mot dans le corpus :

$$tf_{t,d} \times idf_t$$

avec $tf_{t,d}$ le nombre d'occurrences de t dans d

et $idf_t = \log \frac{N}{df_t}$ (N est le nombre total de documents)

In [12]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(X)
```

Out[12]: TfidfVectorizer

TfidfVectorizer

In [13]:
```
import numpy as np
np.set_printoptions(precision=2)

X_TFxIDF = tfidf_vectorizer.transform(X)
print(X_TFxIDF.toarray())
```

```
[[0.39 0.39 0.   0.28 0.28 0.28 0.39 0.39 0.39]
 [0.   0.   0.63 0.45 0.45 0.45 0.   0.   0.  ]]
```

Il existe d'autres systèmes de pondération, en particulier lorsque des classes sont fournies :

- Residual IDF (Rennie and Jaakkola, 2005)
- Odds Ratio (Mladenic and M. Grobelnik, 2009)
- Information Gain (Yang and Pedersen, 1997)
- Chi-squared (Yang and Pedersen, 1997)
- OKAPI BM25 (Robertson et al., 1994)

OKAPI BM25

$$w_{BM25}(t,d) = tf_{BM25}(t,d) \times idf_{BM25}(t)$$

avec :

$$tf_{BM25}(t,d) = \frac{tf(t,d) \times (k_1+1)}{tf(t,d) + k_1 \times (1 - b + b \times dl(d)/dl_{avg})}$$

$$idf_{BM25}(t) = \log \frac{N - df(t) + 0.5}{df(t) + 0.5}$$

où $dl(d)$ = longueur de d, dl_{avg} = longueur moyenne
k_1 et b sont des constates données à priori (en général, $k_1 = 2$ et $b = 0.75$)

Comparaison de deux textes

Les distances usuelles (ex. euclidenne) ne sont pas adaptées.

Dans les espaces à **beaucoup de dimensions** :

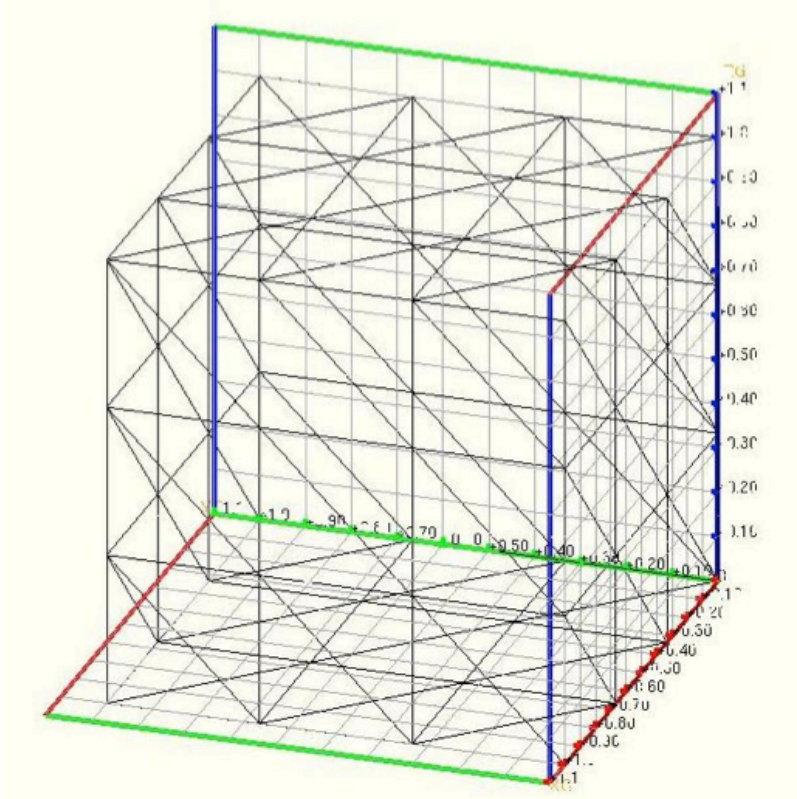
Pourquoi les banquiers n'ont jamais de lingots sphériques ?

Pourquoi les marchands d'oranges occupent beaucoup de place pour empiler peu d'oranges ?

http://www.brouty.fr/Maths/sphere.html (see "Curiosités du calcul")

Malédiction de la dimension (curse of dimensionality)

Richard E. Bellman (1920-1984): les hypervolumes sont presque vides!



Un volume avec $dim = d$ a besoin de 10^d données pour peupler équitablement l'espace.

Produit scalaire et cosinus

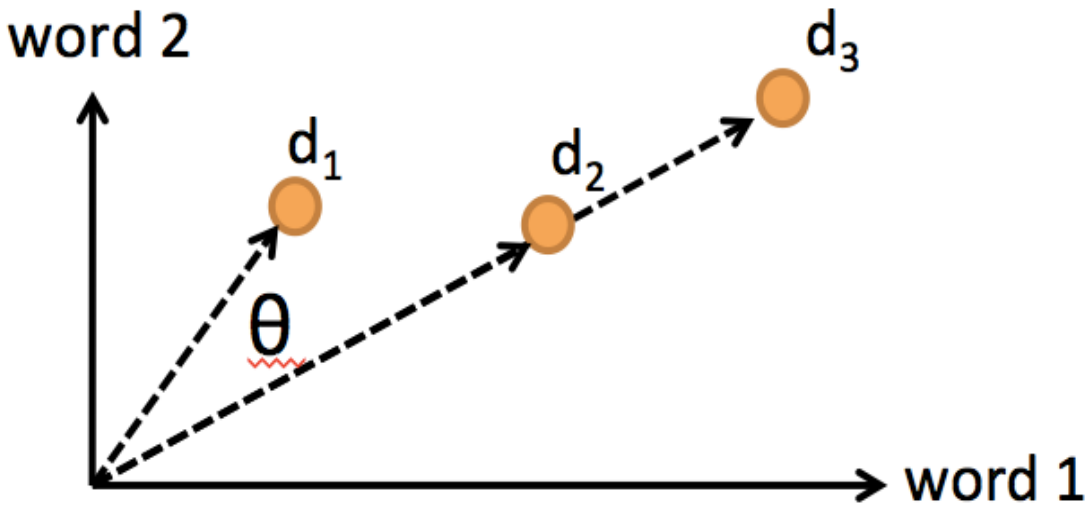
$\vec{x}$ et $\vec{y}$ sont deux vecteurs dans le VSM.

Cosine est une mesure de **similarité** calculée sur l'angle formé par les deux vecteurs :

$$cos(\vec{x}, \vec{y}) = \frac{\vec{x}.\vec{y}}{||\vec{x}||_2 \times ||\vec{y}||_2}$$

Elle prend une valeur entre 0 (rien en commun) et 1 (même vecteurs à une constante près).

Interprétation géométrique



Exemple : distribution des mots les plus fréquents dans Harry Potter

Commençons par lire les données :

In [16]:
```
import os

with open(os.path.join("datasets", "Harry_Potter_1.txt")) as f:
    lines = [line.strip() for line in f.readlines()]

# version de base :
tf_vectorizer = CountVectorizer()

# version avec une liste de mots outils fournie
sw = ["they", "were", "to", "and"] # etc. (il est préférable de charger une liste d
tf_vectorizer = CountVectorizer(stop_words=sw)
# si vous voulez utiliser une liste pré-construite :
#tf_vectorizer = CountVectorizer(stop_words="english")
# si vous voulez utiliser l'encodage TFxIDF :
#tfidf_vectorizer = TfidfVectorizer()

tf_vectorizer.fit(lines)

# montrer l'intégralité du vocabulaire (peut être long) :
#tf_vectorizer.vocabulary_
```

Out[16]:
```
CountVectorizer
CountVectorizer(stop_words=['they', 'were', 'to', 'and']
```

In [17]:
```
# si on veut tester en même temps avec le stemming et la suppression de mots-out

## on charge les mots-outils depuis un fichier :
```

```
#with open(os.path.join("stop", "Stop-words-en.txt")) as f:
#    my_sw_en = [line.strip() for line in f.readlines()]

## analyseur qui réalise le stemming et la suppression des mots-outils
#def stem_sw_words_en(doc):
#    return (stemmer_en.stem(w) for w in analyzer_with_stemming(doc) if w not in my

#stem_sw_vectorizer_en = CountVectorizer(analyzer=stem_sw_words_en)
#stem_sw_vectorizer_en.fit(lines)

#X_hp_stem_sw = stem_sw_vectorizer_en.transform(lines)
#features_hp_stem_sw = stem_sw_vectorizer_en.get_feature_names_out()
#tf_sum_stem_sw = X_hp_stem_sw.sum(axis=0)
#tf_sum_stem_sw = tf_sum_stem_sw.tolist()[0] # conversion en liste

#print_feats(tf_sum_stem_sw, features_hp_stem_sw)
```

In [18]:
```
# puis on construit la matrice Documents x Termes basée sur le vocabulaire

X_hp = tf_vectorizer.transform(lines)
features_hp = tf_vectorizer.get_feature_names_out()
#len(features_hp)
#features_hp[0:10]
```

On peut utiliser quelques fonctions pour afficher le vecteur pour un document en particulier :

In [19]:
```
from scipy.sparse import find, csr_matrix
import pandas as pd

# des options permettent de limiter (ou non) le nombre de lignes/colonnes affichées
# par exemple :
# pd.set_option('display.max_rows', None)

# cette fonction permet d'afficher une "jolie" représentation du vecteur v
# ARGS :
#   v : le vecteur à afficher (par ex. une ligne de la matrice X)
#   features : le vocabulaire
#   top_n : le nombre de mots maximum à afficher
def print_feats(v, features, top_n = 30):
    _, ids, values = find(v)
    feats = [(ids[i], values[i], features[ids[i]]) for i in range(len(list(ids)))]
    top_feats = sorted(feats, key=lambda x: x[1], reverse=True)[0:top_n]
    return pd.DataFrame({"word" : [t[2] for t in top_feats], "value": [t[1] for t i
```

In [20]:
```
print(lines[4])

print_feats(X_hp[4], features_hp, top_n=35)
```

```
Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they we
perfectly normal, thank you very much. They were the last people you'd expect to be
involved in anything strange or mysterious, because they just didn't hold with such
nonsense.
```

Out[20]:

	word	value
0	you	2
1	anything	1
2	be	1
3	because	1
4	didn	1
5	drive	1
6	dursley	1
7	expect	1
8	four	1
9	hold	1
10	in	1
11	involved	1
12	just	1
13	last	1
14	mr	1
15	mrs	1
16	much	1
17	mysterious	1
18	nonsense	1
19	normal	1
20	number	1
21	of	1
22	or	1
23	people	1
24	perfectly	1
25	privet	1
26	proud	1
27	say	1
28	strange	1
29	such	1

	word	value
30	thank	1
31	that	1
32	the	1
33	very	1
34	with	1

Afficher les mots les plus fréquents :

In [21]:
```
n_docs, n_terms = X_hp.shape

# on fait la somme sur toutes les lignes pour chacun des mots
tf_sum = X_hp.sum(axis=0)
tf_sum = tf_sum.tolist()[0] # conversion en liste

print_feats(tf_sum, features_hp)
```

Out[21]:

	word	value
0	the	3627
1	he	1759
2	harry	1326
3	of	1267
4	it	1186
5	was	1186
6	you	1037
7	in	967
8	his	937
9	said	794
10	had	701
11	that	688
12	on	637
13	at	625
14	as	526
15	him	501
16	but	485
17	ron	429
18	with	416
19	all	397
20	what	386
21	out	375
22	up	372
23	for	371
24	hagrid	370
25	be	367
26	them	326
27	there	312
28	have	297
29	hermione	270

Comparer des documents revient à comparer les valeurs de deux colonnes (vecteurs).

In [22]:
```
import math
from scipy.linalg import norm

# fonction calculant le cosinus entre deux vecteurs
def cosinus(i, j):
        # numérateur : <i.j>
    num = i.dot(j.transpose())[0,0]
        # dénominateur : ||i||_2 * ||j||_2
    den = norm(i.todense()) * norm(j.todense())
    if (den>0): # on vérifie que le dénominateur n'est pas nul
        return (num/den)
    else:
        return 0
```

In [23]:
```
print(print_feats(X_hp[5], features_hp, top_n=40))
print(print_feats(X_hp[10], features_hp, top_n=40))
```

	word	value
0	the	4
1	was	4
2	of	3
3	called	2
4	dursley	2
5	had	2
6	he	2
7	in	2
8	neck	2
9	very	2
10	which	2
11	although	1
12	amount	1
13	any	1
14	anywhere	1
15	as	1
16	beefy	1
17	big	1
18	blonde	1
19	boy	1
20	came	1
21	craning	1
22	did	1
23	director	1
24	drills	1
25	dudley	1
26	dursleys	1
27	fences	1
28	finer	1
29	firm	1
30	garden	1
31	grunnings	1
32	hardly	1
33	have	1
34	her	1
35	large	1
36	made	1
37	man	1
38	mr	1
39	mrs	1

	word	value
0	he	2
1	as	1
2	backed	1
3	car	1
4	chortled	1
5	drive	1
6	dursley	1
7	four	1
8	got	1
9	his	1
10	house	1
11	into	1
12	left	1
13	little	1

```
14        mr      1
15    number      1
16        of      1
17       out      1
18       the      1
19      tyke      1
```

In [24]:
```
cosinus(X_hp[5], X_hp[10])
#cosinus(X_hp[5], X_hp[5])
#cosinus(X_hp[5], 3*X_hp[5])
#cosinus(X_hp[2], X_hp[5])
#cosinus(X_hp[2], X_hp[0])
```

Out[24]: 0.2808772073524269

Créer son propre moteur de recherche

Pour créer son propre moteur de recherche maison, la procédure revient à :

- construire un pseudo-document correspondant à la requête, càd un vecteur-requête dans le même espace que les documents
- comparer le vecteur-requête avec tous les vecteurs documents (càd les lignes de la matrice), par ex. avec une mesure cosinus
- trier le vecteur des scores qui en résultent
- afficher les documents qui ont obtenu les meilleurs scores

In [25]:
```
#query = ['privet', 'drive', 'dursley']
#query = ['privet', 'privet', 'drive', 'dursley']
query = ['chess', 'chess', 'chess', 'harry', 'play']
#query = ['1473', 'aaaaarrrgh']

indexes = [np.where(features_hp == q)[0][0] for q in query if q in features_hp]
print(indexes)
```

```
[817, 817, 817, 2245, 3542]
```

On construit un vecteur de la même taille que le vocabulaire. Il est initialisé à zéro, puis on y met la valeur 1 pour les termes de la requête.

De manière alternative, on pourrait mettre un poids aux mots de la requête.

In [26]:
```
query_vec = np.zeros(n_terms)

# pour mettre 1 aux index des mots-clefs
#query_vec[indexes] = 1

# alternative pour pouvoir mettre plus que 1 en répétant les mots-clefs
for tt in indexes:
    query_vec[tt] += 1
```

In [27]:
```
#len(query_vec)
query_vec[3658:3670]
#query_vec[810:830]
#query_vec[0:10]
#query_vec[1410:1420]
```

Out[27]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

On peut vérifier que le vecteur requête contient bien 3 éléments non nuls.

In [28]:
```
query_vec = csr_matrix(query_vec)
query_vec.sum()
```

Out[28]:
```
5.0
```

Calcul du cosinus vis-à-vis de la requête pour 1 document :

In [29]:
```
cosinus(X_hp[4], query_vec)
```

Out[29]:
```
0.0
```

On automatise en calculant pour tous les docs et on triant le résultat :

In [30]:
```
# fonction qui crée un dictionnaire associant le cosinus à chaque document
# puis le trie de manière décroissante

def search(q, X):
    cc = {i: cosinus(X[i], q) for i in range(n_docs)}
    cc = sorted(cc.items(), key=lambda x: x[1], reverse=True)
    return cc
```

In [31]:
```
result = search(query_vec, X_hp)
```

In [32]:
```
result[0:10]
```

Out[32]:
```
[(2137, 0.6154574548966638),
 (3135, 0.30151134457776363),
 (2192, 0.25860327353409435),
 (1060, 0.2461829819586655),
 (2896, 0.24174688920761409),
 (2890, 0.24120907566221092),
 (3179, 0.21320071635561047),
 (129, 0.21320071635561041),
 (815, 0.21320071635561041),
 (942, 0.21320071635561041)]
```

On ne retient que les dix premiers résultats (par exemple).

In [33]:
```
nb_top_docs = 10
top_docs = [r for (r,v) in result[0:nb_top_docs]]
print(top_docs)
```

```
[2137, 3135, 2192, 1060, 2896, 2890, 3179, 129, 815, 942]
```

Pour finir, on peut afficher les textes les plus pertinents pour la requête :

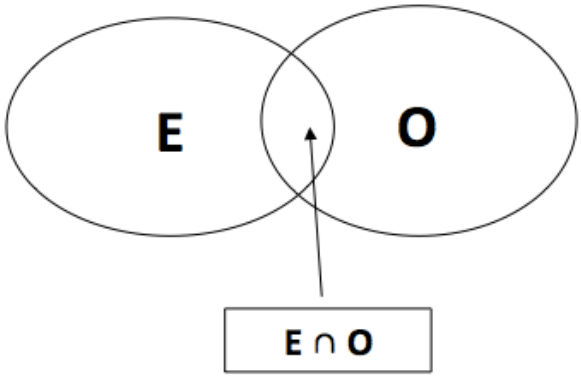
In [34]:
```
for i, td in zip(range(nb_top_docs), top_docs):
    #print(top_feats_in_doc(X_hp, features_hp, td))
    print("%s (%s): %s" % (i+1, td, lines[td]))
```

```
1 (2137): “Want to play chess, Harry?” said Ron.
2 (3135): “Harry!”
3 (2192): The rest of the team hung back to talk to one another as usual at the end
of practice, but Harry headed straight back to the Gryffindor common room, where he
found Ron and Hermione playing chess. Chess was the only thing Hermione ever lost a
t, something Harry and Ron thought was very good for her.
4 (1060): “Harry Potter,” said Harry.
5 (2896): “White always plays first in chess,” said Ron, peering across the board.
“Yes...look...”
6 (2890): Harry and Hermione stayed quiet, watching Ron think. Finally he said, “No
w, don’t be offended or anything, but neither of you are that good at chess —”
7 (3179): “...for the best-played game of chess Hogwarts has seen in many years, I a
ward Gryffindor house fifty points.”
8 (129): Harry groaned.
9 (815): Harry swallowed.
10 (942): “Harry Potter!”
```

Quelques éléments d'évaluation

L'objectif consiste à comparer une "vérité terrain" constituée d'un ensemble ou d'une liste ordonnée de réponses idéales avec la sortie proposée par le moteur de recherche.

Si E = liste attendue des documents pertinents (E pour *Expected*) et O = sortie de l'algorithme de recherche (O pour *Output*) :



précision : P = $|E \cap O|/|O|$

rappel (*recall*) : R = $|E \cap O|/|E|$

F-mesure : FM = $2(P \times R)/(P + R)$