Fouille de textes

L'objectif de ce TP est de se familiariser avec certaines **opérations de base en fouille de textes**. Le logiciel utilisé est nltk (Natural Language ToolKit) qui est basé sur Python.

1 Opérations sur des textes bruts

Peut-être encore davantage que d'autres tâches de fouille de données, la fouille de textes et la recherche de documents posent le problème du prétraitement. Les textes et documents sont en effet généralement disponibles sous des formes et des formats divers : en html, xml, doc, pdf, etc. Dans le cadre limité de ce tp, nous supposerons que les documents ont été traités pour être disponibles sous la forme de corpus exploitables, et nous allons nous focaliser sur des opérations d'analyse globale et de redescription.

Commençons par invoquer nltk et les librairies utiles :

```
In [4]: import nltk
In [5]: import numpy
In [6]: import matplotlib
```

Nous chargeons ensuite un texte sur lequel nous ferons diverses opérations.

```
In [7]: from nltk.book import *
```

Cette commande charge à la fois 9 textes « classiques » et les fonctions prédéfinies de nltk.

Le text2 est celui du roman *Sense and sensibility* de Jane Austen, écrit en 1811.

1.1 Comptages et fréquences

Nous commençons par quelques **opérations élémentaires** : calcul de la *longueur du texte* en mots, *taille du vocabulaire* (nombre de mots différents), *diversité du vocabulaire* (rapport de la longueur du texte sur la taille du vocabulaire).

Calcul du **nombre de mots** de *Sense and Sensibility* :

```
In [8]: len(text2)
```

Calcul de la **taille du vocabulaire** utilisé (pour cela on calcule d'abord l'ensemble des mots par la fonction set):

```
In [9]: len(set(text2))
```

Nous allons maintenant définir une fonction de calcul de la **diversité lexicale**.

```
def lexical_diversity(text):
    return len(text) / len(set(text))
```

Appliquez la fonction à *Sense and Sensibility*. Qu'obtenez-vous ?

Appliquez cette même fonction au text1 (*Moby Dick* de Herman Melville en 1851) et au text6 (*Monty Python and the Holy Grail*, film britannique sorti en 1975 et écrit et réalisé par Terry Gilliam et Terry Jones des Monty Python). Faut-il en conclure quelque chose ?

Il est également possible de compter le nombre de fois où un mot apparaît dans un texte. Par exemple (ici on applique une méthode 'count' à un objet, le texte) :

```
In [10]: text2.count("love")
```

```
Out[10]: 77
```

Mais :

```
In [13]: text6.count("love")
Out[13]: 0
```

Nous poursuivons avec quelques **opérations statistiques** : *fréquence de chaque mot, courbe cumulative de fréquence*.

La fonction FreqDist définie dans nltk permet de calculer la fréquence de chaque mot du texte qui lui est passé en argument :

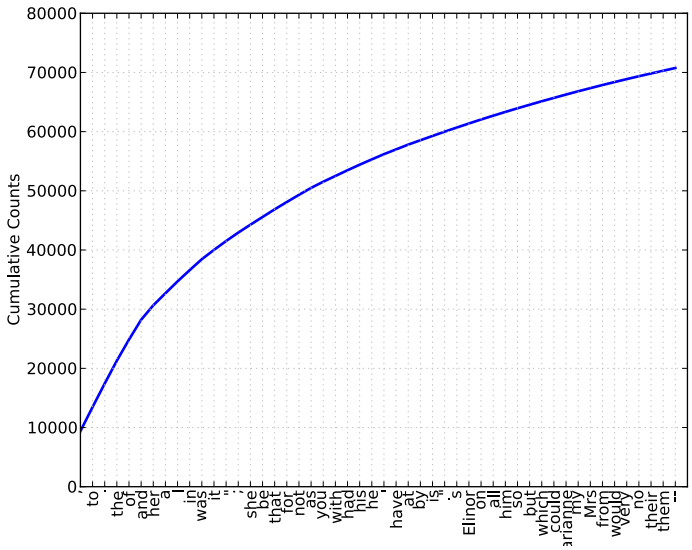
```
In [16]: fdist2 = FreqDist(text2)
In [17]: vocabulary2 = fdist2.keys()
In [18]: vocabulary2[:20]
Out[18]: [',', 'to', '.', 'the', 'of', 'and', 'her', 'a', 'I', 'in', 'was', 'it', '"', ';', 'she', 'be', 'that', 'for', 'not', 'as']
```

L'expression fdist2.keys() a pour valeur l'ensemble des mots dans le tableau associatif fdist2.

Ici on a demandé les 20 mots (dont des signes de ponctuation) les plus fréquents dans *Sense and Sensibility*.

On peut également tracer la courbe cumulée du nombre de fois où les mots les plus fréquents sont utilisés :

```
In [19]: fdist2.plot(50, cumulative=True)
```



Calculons maintenant l'ensemble des mots de longueur supérieure à 7 caractères et apparaissant plus de 50 fois dans *Sense and Sensibility* :

```
In [26]: sorted([w for w in set(text2) if len(w) > 7 and fdist2[w] > 50])
Out[26]:
['Dashwood', 'Jennings', 'Marianne', 'Middleton', 'Willoughby', 'acquaintance', 'affection', 'attention', 'behaviour', 'continued', 'engagement', 'feelings', 'happiness', 'immediately', 'pleasure', 'returned', 'situation', 'something', 'therefore', 'together']
```

On peut vouloir calculer la fréquence d'utilisation des mots selon leur longueur :

```
In [27]: fdist22 = FreqDist([len(w) for w in text2])
In [28]: fdist22.keys()
Out[28]: [3, 2, 1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 17, 16]
```

Les mots les plus fréquents dans *Sense and Sensibility* sont donc de longueur 3, mais combien y en a-t-il ?

```
In [29]: fdist22.items()
Out[29]:
[(3, 28839), (2, 24826), (1, 23009), (4, 21352), (5, 11438), (6, 9507), (7, 8158), (8, 5676), (9, 3736), (10, 2596), (11, 1278), (12, 711), (13, 334), (14, 87), (15, 24), (17, 3), (16, 2)]
```

Il y a donc 28839 mots de longueur 3, alors qu'il y a 2 mots seulement de longueur 16.

Mais lesquels sont ces deux mots ? **Ecrivez l'expression Python permettant de les identifier**.

Que remarquez-vous ? Avez-vous eu la réponse que vous attendiez ? **Modifiez l'expression python** pour calculer la réponse attendue. Refaites le pour les mots de longueur 17.

Calculez maintenant le nombre de mots de longueur 3 *différents* dans *Sense and Sensibility*.

Il existe également des fonctions permettant d'**exprimer des conditions sur les mots**. Par exemple, on peut calculer les mots se terminant par 'ableness' :

```
In [38]: sorted([w for w in set(text2) if w.endswith('ableness')])
Out[38]: ['companionableness', 'reasonableness', 'unsuitableness']
```

1.2 Comptage de mots dans différents documents

Nous allons comparer des documents en fonction de la fréquence de mots dans chacun d'eux.

On charge d'abord le corpus Brown qui contient des extraits de journaux et revues, et on met dans news_text la liste des mots rencontrés. On va alors **compter le nombre d'occurrences de mots modaux**.

```
In [132]: from nltk.corpus import brown
In [133]: news_text = brown.words(categories='news')
In [135]: fdist = nltk.FreqDist([w.lower() for w in news_text])
In [136]: modals = ['can', 'could', 'may', 'might', 'must', 'will']
In [137]: for m in modals:
   .....:     print m + ':', fdist[m]
   .....:
can: 94   could: 87   may: 93   might: 38   must: 53   will: 389
```

Faites la même chose avec la catégorie 'science_fiction' (au lieu de 'news') et avec les mots commençant par wh (*what, where, who, why, …*).

Nous allons maintenant faire une comparaison systématique entre catégories sur les mots modaux.

```
In [138]: cfd = nltk.ConditionalFreqDist(
   .....: (genre, word)
   .....: for genre in brown.categories()
   .....: for word in brown.words(categories = genre))
In [139]:
In [139]: genres = ['news', 'religion', 'hobbies', 'science_fiction', 'romance', 'humor']
In [140]: cfd.tabulate(conditions = genres, samples = modals)
                  can could  may might must will
        news       93   86    66    38    50   389
    religion       82   59    78    12    54    71
```

```
      hobbies   268   58   131   22    83   264
science_fiction  16   49     4   12     8    16
      romance   74  193    11   51    45    43
        humor   16   30     8    8     9    13
```

Quel est le mot modal le plus fréquent dans la catégorie 'news', et dans la catégorie 'romance' ? Pouvait-on s'y attendre ?

Ré-essayez avec les mots en *wh*.

Pourrait-on distinguer les différentes catégories en utilisant la fréquence des mots ?

1.3 Utilisation de bigrams pour générer automatiquement du texte

1.4 Les stopwords

On va d'abord examiner quels sont les mots les moins fréquents dans *Sense and Sensibility*. Pour cela, on calcule les mots présents dans ce roman et on en **retire tous les mots usuels en anglais**. Ceux-ci sont obtenus grâce à un corpus spécial de nltk. Voici donc la fonction qui fait ce travail :

```
def unusual_words(text):
    text_vocab = set(w.lower() for w in text if w.isalpha())
    # on ne considere que les mots et on enleve les majuscules
    english_vocab = set(w.lower() for w in nltk.corpus.words.words())
    unusual = text_vocab.difference(english_vocab)
    return sorted(unusual)
```

Ce qui donne :

```
>>> unusual_words(nltk.corpus.gutenberg.words('austen-sense.txt'))
['abbeyland', 'abhorrence', 'abominably', 'abridgement', 'accordant', 'accustomary',
'adieus', 'affability', 'affectedly', 'aggrandizement', 'alighted', 'allenham',
'amiably', 'annamaria', 'annuities', 'apologising', 'arbour', 'archness', ...]
```

Essayez sur le corpus de 'chat' :

```
>>> unusual_words(nltk.corpus.nps_chat.words())
```

Il existe un **corpus de stopwords** disponible dans nltk. Il comporte 127 mots.

```
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
['a', "a's", 'able', 'about', 'above', 'according', 'accordingly', 'across',
'actually', 'after', 'afterwards', 'again', 'against', "ain't", 'all', 'allow',
'allows', 'almost', 'alone', 'along', 'already', 'also', 'although', 'always', ...]
```

Nous allons définir une fonction qui calcule la fraction de mots d'un texte qui ne font pas partie des stopwords. Nous l'appliquons alors au corpus des dépêches Reuters.

```
>>> def content_fraction(text):
...     stopwords = nltk.corpus.stopwords.words('english')
```

```
...     content = [w for w in text if w.lower() not in stopwords]
...     return len(content) / len(text)
...
>>> content_fraction(nltk.corpus.reuters.words())
0.6599769539328526
```

Quelle est cette proportion dans *Sense and Sensibility* ?

2 Sélection de descripteurs pertinents

2.1 Utilisation de WordNet

Nous allons maintenant utiliser le réseau sémantique WordNet pour étudier les relations entre mots.

```
>>> from nltk.corpus import wordnet as wn
>>> wn.synsets('motorcar')
[Synset('car.n.01')
```

Le mot 'motorcar' n'a qu'un sens dans WordNet comme le montre le fait que l'ensemble des synonymes a un seul élément. Qu'en est-il avec le mot 'dish' ?

On peut examiner les différents noms correspondant à l'unique sens de 'motorcar' :

```
>>> wn.synset('car.n.01').lemma_names
['car', 'auto', 'automobile', 'machine', 'motorcar']
```

Combien y a-t-il de mots dans WordNet correspondant au premier synonyme de 'dish', et combien pour le premier sens en tant que verbe ?

On peut obtenir la définition d'un sens du mot. De même, on peut demander un exemple d'usage :

```
>>> wn.synset('car.n.01').definition
'a motor vehicle with four wheels; usually propelled by an internal combustion engine'
>>> wn.synset('car.n.01').examples
['he needs a car to get to work']
```

Quelle est la définition du 3ème sens de 'dish' en tant que nom, et quel exemple est fourni ?

Nous allons maintenant nous intéresser aux sous-classes d'un mot (ses **hyponymes**) et à ses sur-classes (ses hypernymes).

Sélectionnons le 1er sens du mot 'motorcar', puis cherchons ses hyponymes dans WordNet, avant d'en donner les mots (lemmas) qui les décrivent :

```
>>> motorcar = wn.synset('car.n.01')
>>> types_of_motorcar = motorcar.hyponyms()
>>> types_of_motorcar[26]
Synset('ambulance.n.01')
>>> sorted([lemma.name for synset in types_of_motorcar for lemma in synset.lemmas])
['Model_T', 'S.U.V.', 'SUV', 'Stanley_Steamer', 'ambulance', 'beach_waggon',
'beach_wagon', 'bus', 'cab', 'compact', 'compact_car', 'convertible',
```

```
'coupe', 'cruiser', 'electric', 'electric_automobile', 'electric_car',
'estate_car', 'gas_guzzler', 'hack', 'hardtop', 'hatchback', 'heap',
'horseless_carriage', 'hot-rod', 'hot_rod', 'jalopy', 'jeep', 'landrover',
'limo', 'limousine', 'loaner', 'minicar', 'minivan', 'pace_car', 'patrol_car',
'phaeton', 'police_car', 'police_cruiser', 'prowl_car', 'race_car', 'racer',
'racing_car', 'roadster', 'runabout', 'saloon', 'secondhand_car', 'sedan',
'sport_car', 'sport_utility', 'sport_utility_vehicle', 'sports_car', 'squad_car',
'station_waggon', 'station_wagon', 'stock_car', 'subcompact', 'subcompact_car',
'taxi', 'taxicab', 'tourer', 'touring_car', 'two-seater', 'used-car', 'waggon',
'wagon']
```

Faites de même avec 'dish'.

Il est facile d'obtenir de la même manière les **hypernymes** :

```
>>> motorcar.hypernyms()
[Synset('motor_vehicle.n.01')]
```

Il est intéressant d'examiner l'ensemble des **chemins ascendants vers la racine de l'ontologie** WordNet. Dans le cas de 'motorcar', il existe deux chemins distincts :

```
>>> paths = motorcar.hypernym_paths()
>>> len(paths)
2
>>> [synset.name for synset in paths[0]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',
'instrumentality.n.03', 'container.n.01', 'wheeled_vehicle.n.01',
'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
>>> [synset.name for synset in paths[1]]
['entity.n.01', 'physical_entity.n.01', 'object.n.01', 'whole.n.02', 'artifact.n.01',
'instrumentality.n.03', 'conveyance.n.03', 'vehicle.n.01', 'wheeled_vehicle.n.01',
'self-propelled_vehicle.n.01', 'motor_vehicle.n.01', 'car.n.01']
```

Qu'en est-il de 'dish' ?

L'existence de WordNet permet de **calculer des distances sémantiques entre mots**. Par exemple, on peut considérer que deux mots sont d'autant plus proches sémantiquement qu'ils partagent un hyponyme à distance faible au-dessus d'eux.

Cherchons par exemple la similarité entre les mots 'right_whale', 'orca', 'minke', 'tortoise' et 'novel'.

On commence par charger le premier sens de chacun de ces mots :

```
>>> right = wn.synset('right_whale.n.01')
>>> orca = wn.synset('orca.n.01')
>>> minke = wn.synset('minke_whale.n.01')
>>> tortoise = wn.synset('tortoise.n.01')
>>> novel = wn.synset('novel.n.01')
```

On peut ensuite chercher **l'hypernyme commun le plus bas dans l'ontologie** :

```
>>> right.lowest_common_hypernyms(minke)
[Synset('baleen_whale.n.01')]
>>> right.lowest_common_hypernyms(orca)
[Synset('whale.n.02')]
>>> right.lowest_common_hypernyms(tortoise)
[Synset('vertebrate.n.01')]
>>> right.lowest_common_hypernyms(novel)
[Synset('entity.n.01')]
```

Il est clair que 'right_whale' et 'whale' sont proches, ce qui n'est pas le cas de 'right_whale' et de 'novel' dont l'hypernyme commun le plus spécifique est 'entity' qui est la racine de WordNet. On peut quantifier cette proximité en regardant **la profondeur du premier hypernyme commun**.

```
>>> wn.synset('baleen_whale.n.01').min_depth()
14
>>> wn.synset('whale.n.02').min_depth()
13
>>> wn.synset('vertebrate.n.01').min_depth()
8
>>> wn.synset('entity.n.01').min_depth()
0
```

Une mesure découlant de cette « distance arborée » est calculée par la fonction path_similarity. Elle retourne un nombre entre 0 et 1, 0 indiquant une similarité nulle alors que 1 est la similarité entre un terme et lui-même (la fonction retourne -1 si aucun chemin n'est trouvé entre les deux termes (e.g. entre un nom et un verbe).

```
>>> right.path_similarity(minke)
0.25
>>> right.path_similarity(orca)
0.16666666666666666
>>> right.path_similarity(tortoise)
0.076923076923076927
>>> right.path_similarity(novel)
0.043478260869565214
```

Quel degré de similarité trouvez-vous entre 'novel', 'romance', 'news' et 'report' ?

2.2 Stemming et lemmatisation

Souvent il est intéressant de ramener un texte à des mots racines afin d'éliminer les différences dues à des formes particulières des mots. Un procédé pour ce faire est le **stemming**. Il n'existe pas de manière unique de procéder et plusieurs « stemmers » sont donc disponibles.

Partons par exemple du texte suivant :

```
>>> raw = """DENNIS: Listen, strange women lying in ponds distributing swords
... is no basis for a system of government.  Supreme executive power derives from
... a mandate from the masses, not from some farcical aquatic ceremony."""
```

```
>>> tokens = nltk.word_tokenize(raw)
```

L'utilisation du stemmer de Porter, l'un des plus utilisés, conduira à :

```
>>> porter = nltk.PorterStemmer()
>>> [porter.stem(t) for t in tokens]
['DENNI', ':', 'Listen', ',', 'strang', 'women', 'lie', 'in', 'pond',
'distribut', 'sword', 'is', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
'.', 'Suprem', 'execut', 'power', 'deriv', 'from', 'a', 'mandat', 'from',
'the', 'mass', ',', 'not', 'from', 'some', 'farcic', 'aquat', 'ceremoni', '.']
```

Mais le stemmer de Lancaster produit :

```
>>> lancaster = nltk.LancasterStemmer()
>>> [lancaster.stem(t) for t in tokens]
['den', ':', 'list', ',', 'strange', 'wom', 'lying', 'in', 'pond', 'distribut',
'sword', 'is', 'no', 'bas', 'for', 'a', 'system', 'of', 'govern', '.', 'suprem',
'execut', 'pow', 'der', 'from', 'a', 'mand', 'from', 'the', 'mass', ',', 'not',
'from', 'som', 'farc', 'aqu', 'ceremony', '.']
```

La **lemmatisation** ne supprime les suffixes que si le mot résultant fait partie du dictionnaire. Par exemple :

```
>>> wnl = nltk.WordNetLemmatizer()
>>> [wnl.lemmatize(t) for t in tokens]
['DENNIS', ':', 'Listen', ',', 'strange', 'woman', 'lying', 'in', 'pond',
'distributing', 'sword', 'is', 'no', 'basis', 'for', 'a', 'system', 'of',
'government', '.', 'Supreme', 'executive', 'power', 'derives', 'from', 'a',
'mandate', 'from', 'the', 'mass', ',', 'not', 'from', 'some', 'farcical',
'aquatic', 'ceremony', '.']
```

2.3 Tagging

Afin de distinguer entre les différents sens possibles d'un mot, il est intéressant de chercher leur rôle grammatical dans le texte étudié. C'est en particulier la tâche du Part-Of-Speech tagging.

```
>>> text = nltk.word_tokenize("And now for something completely different")
>>> nltk.pos_tag(text)
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'),
('completely', 'RB'), ('different', 'JJ')]
```

Le mot *'and'* est donc reconnu comme CC, c'est-à-dire une conjonction. Les mots *'now'* et *'completely'* comme des RB ou adverbes, *'for'* est IN, c'est-à-dire une préposition, *'something'* est NN, soit un nom et *'different'* est JJ, c'est-à-dire un adjectif.

Considérons un cas plus difficile incluant des homonymes :

```
>>> text = nltk.word_tokenize("They refuse to permit us to obtain the refuse permit")
>>> nltk.pos_tag(text)
```

```
[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'),
('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]
```

On voit par exemple que le mot '*permit*' est bien reconnu comme un verbe, VB, et comme un nom, NN, de manière appropriée dans le texte. Il en est de même pour '*refuse*'.

On peut aussi chercher les mots qui ont le même comportement grammatical dans un document. Ici, la méthode text.similar() prnd en entrée un mot *w*, recherche tous les contextes w_1ww_2 dans le document analysé et retourne tous les mots *w'* partageant le même contexte $w_1w'w_2$.

```
>>> text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())
>>> text.similar('woman')
Building word-context index...
man time day year car moment world family house country child boy
state job way war girl place room word
>>> text.similar('bought')
made said put done seen had found left given heard brought got been
was set told took in felt that
>>> text.similar('over')
in on to of and for with from at by that into as up out down through
is all about
>>> text.similar('the')
a his this their its her an that our any all one these my in your no
some other and
```

2.4 Sélection de mots pertinents par tf-idf

L'analyse tf-idf permet de caractériser la pertinence de mots pour distinguer des textes pris dans un corpus. Un mot est ainsi d'autant plus pertinent pour distinguer un texte au sein d'un corpus que sa fréquence dans ce texte est différente de la fréquence moyenne dans le corpus.

Les fonctions tf, idf et tfidf sont définis dans nltk. Nous allons considérer trois textes du corpus book de nltk.

```
In [46]: from nltk.book import *
In [47]: text1
Out[47]: <Text: Moby Dick by Herman Melville 1851>
In [66]: mytexts = nltk.TextCollection([text1, text2, text3])
```

Calculons maintenant la fréquence du mot 'is' dans les trois textes :

```
In [77]: mytexts.tf('is',text1)
Out[77]: 0.0064987596762505796
In [78]: mytexts.tf('is',text2)
Out[78]: 0.0051421144826806807
In [79]: mytexts.tf('is',text3)
Out[79]: 0.0059646144223036365
```

Calculons l'*inverse document frequency* du mot 'companionableness' dans nos trois textes :

```
In [85]: mytexts.idf('companionableness')
Out[85]: 1.0986122886681098
```

Et **le score tf-idf** de ce mot :

```
In [86]: mytexts.tf_idf('companionableness', text2)
Out[86]: 7.7598765939715046e-06
```

On va maintenant chercher les mots les plus pertinents pour caractériser un texte. Pour cela on va ordonner les mots en fonction de leur score tf-idf pour ce document (et ce corpus).

Nous définissons une fonction prenant en entrée une liste de mots, un texte et un corpus de textes et retournant la liste ordonnée de ces mots en fonction du score tf-idf décroissant.

```
def sort_by_tfidf(words, document, documents):
    l = []
    for word in words:
        l.append((documents.tf_idf(word, document), word))

    l.sort(reverse = True)

    res = []
    for tfidf, word in l:
        res.append(word)
    return res
```

On peut alors appeler cette fonction, par exemple pour trouver les mots (et signes de ponctuation) les plus pertinents pour distinguer le text2 (‘*Sense and Sensibility*’) dans notre corpus de trois textes.

```
In [129]: maliste2 = fcts.sort_by_tfidf(set(text2), text2, mytexts)
In [131]: maliste2[:20]
Out[131]: ['Elinor', 'Marianne', '"', '."', 'Dashwood', 'Jennings', 'Miss', 'Mrs', 'Lucy', 'Colonel', '--', ',"',
'Brandon', '-', 'Ferrars', 'Middleton', 'Lady', 'Edward',  'Barton',  'Willoughby']
```

Que pensez-vous de ces mots ?

Nous allons maintenant examiner une liste de quelques mots :

```
In [270]: somewords = ['are', 'we', 'inside', 'neighbours', 'love', 'murder', 'honor']
```

Et voir et les ordonner selon leur score tf-idf par rapport aux 9 textes classiques du corpus Gutenberg. Par exemple, pour le text8 qui correspond à ‘Hamlet’ de Shakespeare :

```
In [271]: fcts.sort_by_tfidf(set(somewords), text8, mytexts)
Out[272]: ['murder', 'honor', 'we', 'neighbours', 'love', 'inside', 'are']
```

Est-ce une surprise ? Que prévoyez-vous pour le texte ‘Sense and Sensibility’ ? Que donne le système ?

Essayez avec d’autres textes, comme la Bible par exemple.