Traitement Automatique de la Langue Naturelle

Introduction à NLTK et analyse lexicale et syntaxique de corpus

François Bouchet[1]

[1] LIP6 / SU

2018-2019

M2 ANDROIDE – EVIJV

Plan du cours

Toolkits et outils pour le TALN

Objectif : ne pas recoder tous les traitements de base utiles à l'analyse de texte

- GATE (Java, LGPL): General Architecture for Text Engineering
- LinguaStream (Java, gratuit pour recherche) – Université de Caen
- MontyLingua (Python|Java, gratuit pour recherche) – MIT
- NLTK (Python, Apache 2.0)
- OpenNLP (Java, Apache 2.0)
- natural (JavaScript, libre)
- NooJ (.NET framework|Java, gratuit pour recherche) – Université de Franche-Comté
- UniteX (Java|C++, LGPL) – LADL (Univ. Paris 7)

Pourquoi Python ?

- Apprentissage rapide : syntaxe et sémantique transparente
- Capacité à traiter les chaînes de caractères de manière simple
- Langage interprété : facilités d'exploration
- Langage orienté objet : encapsulation et réutilisabilité du code écrit
- Typage dynamique pour développement rapide
- De nombreuses bibliothèques : traitement numérique, connectivité web. . . et traitement de la langue naturelle, **NLTK**.

Quel environnement ?

- Python 2.6+/3.x
- NLTK 2.0
- NLTK-Data : corpus de données sur lesquels travailler
- NumPy : bibliothèque de calcul scientifique, utile pour certaines tâches d'analyse linguistique
- Matplotlib : bibliothèque pour la visualisation de données (graphiques linéaires, diagrammes en bâtons. . .)

NLTK : présentation

- Créé en 2001 (UPenn)
- Avantages :
  - **Simple :** approprié pour l'apprentissage du TAL par la pratique, sans devoir gérer les sous-tâches mineures qui prennent du temps
  - **Consistant :** noms de méthodes claires, structure de données et interfaces consistantes
  - **Extensible :** facile de proposer des implémentations alternatives pour accomplir une tâche
  - **Modulaire :** possibilité d'utiliser une partie du toolkit sans comprendre le reste
- Inconvénients :
  - Non encyclopédique - mais les outils principaux sont là
  - Pas d'optimisation poussée (algorithmes complexes ou astuces de programmation peu lisibles)

NLTK : installation

Prérequis : Python 2.5+ pour NLTK 2.0 / Python 3.3+ pour NLTK 3.x

Sur une machine avec les droits d'administrateurs :

- Installer easy_install en téléchargeant puis exécutant : `https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py`.
- Installer Numpy (utile pour certaines fonctions de NLTK) :
  – sous Windows, en téléchargeant puis en exécutant : `http://sourceforge.net/projects/numpy/files/latest/download`
  – sous Linux : `pip install -U numpy`
- Installer NLTK : `pip install -U pyyaml nltk`

NLTK : installation

Prérequis : Python 2.5+ pour NLTK 2.0 / Python 3.3+ pour NLTK 3.x

Sur les machines de SU :

- Normalement, déjà installé :
  - `python`
  - `import nltk`
- Sinon :
  - Récupérer l'archive `http://fbouchet.vorty.net/classes/evijv/2017/nltk.tar.gz`
  - Décompresser dans un dossier local (*e.g.* ∼/EVIJV)
  - Ajouter les packages contenus au PYTHONPATH : `export PYTHONPATH="$PYTHONPATH:/fullPath/votreNom/EVIJV/"`

Commençons avec Python

Lancer l'interpréteur python : `python`

```
>>> 2+2
4
```

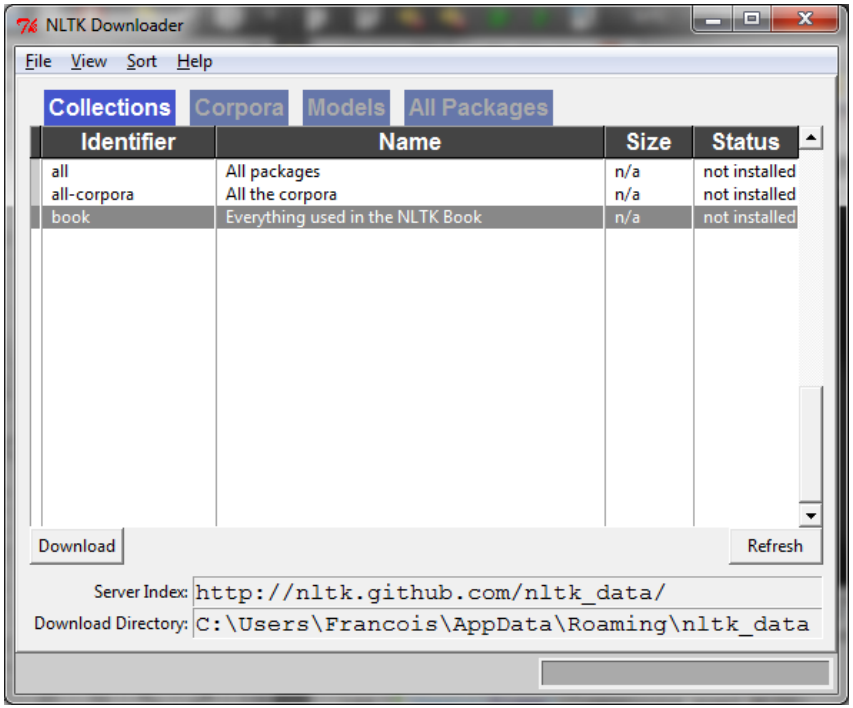
Tester quelques erreurs :

```
>>> 2+
File "<stdin>", line 1, in <module>
SyntaxError:  invalid syntax
>>> 2/0
File "<stdin>", line 1, in <module>
ZeroDivisionError:  division by zero
```

Commençons avec NLTK

Charger la bibliothèque NLTK (préalablement installée) :

```
>>> import nltk
>>> nltk.download()
```



Télécharger la collection “book” si elle n’est pas installée

Plan du cours

Corpus

- Un **corpus** est un ensemble de textes ou énoncés linguistiques.
- En TAL, un corpus permet :
  - de mesurer des tendances, extraire des n-grammes, *etc.*
  - de valider des approches/hypothèses dans un contexte donné
- Dans le cadre d'une application de TAL, le corpus doit être *représentatif* du domaine de langue qui sera traité.
- Collecter un corpus est une tâche qui peut être longue et coûteuse

Corpus dans NLTK

NLTK est distribué avec plusieurs corpus. Pour charger un corpus de livres (en anglais) :

```
>>> from nltk.book import *
```

Il est maintenant possible d'utiliser `text1` à `text9` :

```
>>> text1
<Text:  Moby Dick by Herman Melville 1851>
>>> text3
<Text:  The Book of Genesis>
>>> text5
<Text:  Chat corpus>
```

Concordancier

Recherche d'un mot dans un corpus texte avec le contexte associé :

```
>>> textToSearch.concordance(searchedWord)
```

Exercice

Chercher les occurences des mots "great" et "god" dans les textes 1 (Moby Dick - MD), 3 (La Genèse - Gen) et 5 (Chat).

- combien de fois apparaissent-ils ?
- que penser de leur usage ?

Concordancier

Recherche d'un mot dans un corpus texte avec le contexte associé :

```
>>> textToSearch.concordance(searchedWord)
```

Exercice

Chercher les occurences des mots "great" et "god" dans les textes 1 (Moby Dick - MD), 3 (La Genèse - Gen) et 5 (Chat).

- combien de fois apparaissent-ils ?
  "great": MD (306), Gen (32), Chat (21)
  "god": MD (152), Gen (231), Chat (7)
- que penser de leur usage ?
  Très différent, *e.g.* God est souvent suivi d'un verbe dans Gen et souvent précédé par "thank" dans Chat

Mots similaires

Chercher les mots qui apparaissent dans un même contexte :

```
>>> textToSearch.similar(searchedWord)
```

Chercher un contexte similaire dans lequel N mots apparaissent :

```
>>> textToSearch.common_contexts([word1, word2])
```

Exemple

```
>>> text1.similar("great")
good long sea vast whale whole dead large living other
small last mighty more much same ...
>>> text1.common_contexts(["great", "small"])
a_whale how_the so_a the_fish too_a
```

Diversité du vocabulaire

Différences importantes selon les corpus : *comment les mesurer ?*

- Taille du corpus :
```
>>> len(text1)
260819
>>> len(text3)
44764
```
- Compter le nombre d'élements linguistiques :
```
>>> len(set(text1))
19317
>>> len(set(text3))
2789
```
- Visualiser les éléments linguistiques :
```
>>> [sorted(set(text1))[:20]]
['!', "'", '(', ')', ..., 'A', 'Abel', ...,
'Abimelech']
```

Diversité du vocabulaire

- Évaluer la diversité lexicale :

```
>>> def div_lex(text):
...     return 1.0 * len(text) / len(set(text))
>>> div_lex(text1) #mobydick
13.50
>>> div_lex(text3) #genese
14.94
>>> div_lex(text5) #chat
7.42
```

Distribution fréquentielle : objectifs

Quels sont les mots les plus représentatifs du sujet ou du genre d'un corpus ?

Applications possibles :

- Classification automatique de type d'articles
- Identification automatique d'un locuteur
- Identification de la difficulté d'un texte
- Analyse de sentiments
- ...

Exemples suivants sur text1 : testez un autre corpus !

Distribution fréquentielle – hypothèse 1

Quels sont les mots les plus représentatifs du sujet ou du genre d'un corpus ?

H1 : Les mots les plus fréquents ?

```
>>> fdist1 = FreqDist(text1)
>>> fdist1
<FreqDist with 19317 samples and 260819 outcomes>
```

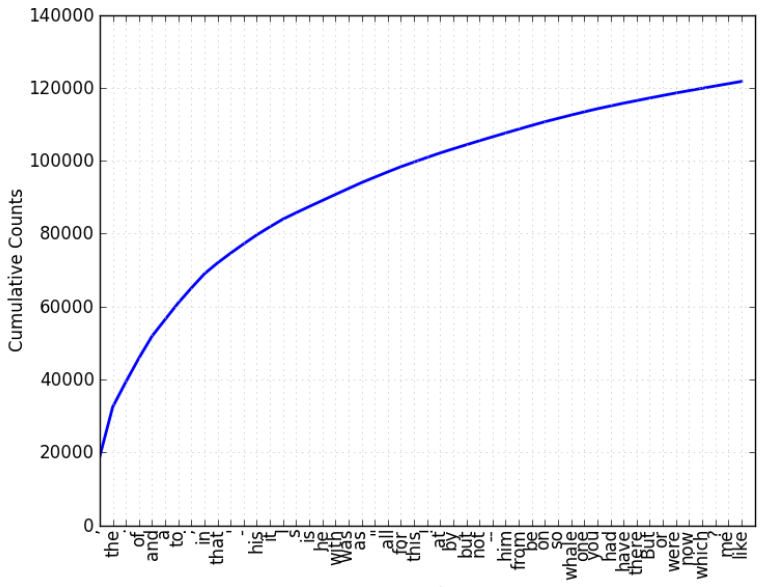
FreqDist est un *dictionnaire* :

```
>>> vocabulary1 = fdist1.keys()
>>> vocabulary1[:10]
[',', 'the', '.', 'of', 'and', 'a', 'to', ';', 'in',
'that']
>>> fdist1['the']
13721
```

Distribution fréquentielle – hypothèse 1

Que représentent les mots les plus fréquents du corpus ?

```
>>> fdist1.plot(50, cumulative=True)
```



50 mots représentent 46% du livre !

Loi de Zipf : fréquence d'un mot $f(n) = K/n$

Distribution fréquentielle – hypothèse 2

Quels sont les mots les plus représentatifs du sujet ou du genre d'un corpus ?

H2 : Les mots les moins fréquents ?

```
>>> hap1=fdist1.hapaxes()
>>> len(hap1)
9002
>>> import random
>>> random.shuffle(hap1)
>>> hap1[:5]
['adjust', 'lt', 'decidedly', 'Europa', 'Murray']
```

Hapax : mot n'apparaissant qu'une seule fois dans un corpus

Distribution fréquentielle – hypothèses 3 et 4

Quels sont les mots les plus représentatifs du sujet ou du genre d'un corpus ?

H3 : Les mots les plus longs ?

```
>>> long1 = [w for w in set(text1) if len(w)>12]
>>> sorted(long1)[:5]
['Archipelagoes', 'Bibliographical', 'CIRCUMNAVIGATION',
'CONVERSATIONS', 'Circumnambulate']
```

H4 : Les mots les plus longs et les plus fréquents ?

```
>>> longfreq1 = [w for w in set(text1) if len(w)>12 and
fdist1[w]>5]
>>> sorted(longfreq1)[:5]
['Mediterranean', 'circumfererence', 'circumstances',
'comparatively', 'conscientious']
```

Distribution fréquentielle – hypothèse 4

Les mots de plus de 10 caractères apparaissant plus de 5 fois :

Moby Dick	**Genese**	**Chat**
circumstances	Peradventure	#14-19teens
significance	circumcised	)))))))))
encountering	peradventure	#talkcity_adults
Nevertheless	buryingplace	Compliments
superstitious	everlasting	(((((((((
accomplished	generations	
contrivances	ringstraked	
representing	exceedingly	
considerably	Philistines	
impressions		
. . . (92 more)		

Distribution fréquentielle – hypothèse 5

H5 : s'intéresser aux séquences de mots

Collocation : séquence de mots qui occurent ensemble

N-grammes : ensemble de N mots (*e.g.* bigrammes, trigrammes)

```
>>> text1.collocations()
Building collocations list
Sperm Whale; Moby Dick; White Whale; old man; Captain
Ahab; sperm whale; Right Whale; Captain Peleg; New
Bedford; Cap Horn; cried Ahab; years ago; lower jaw; never
mind; Father Mapple; cried Stubb; chief mate; white whale;
ivory leg; one hand
```

Distribution fréquentielle – opérations

Autres opérations possibles sur une distribution fréquentielle `fdist` :

`fdist = FreqDist(txt)`	Création d'une distribution fréquentielle
`fdist[mot]`	Nombre de fois où le mot apparaît
`fdist.freq(mot)`	Fréquence (%) du mot dans le corpus
`fdist.N()`	Nombre total d'expressions linguistiques
`fdist.keys()`	Mots par ordre décroissant
`fdist.max()`	Mot le plus courant dans le corpus
`fdist.plot()`	Affiche la distribution fréquentielle

Plan du cours

Corpus composites

- Jusque lors : un corpus = 1 texte
- Maintenant : un corpus = plusieurs textes similaires ou distincts selon des critères connus, que l'on peut considérer ensemble ou séparément

Quelques corpus (1/2) : Brown

- `nltk.corpus.brown`
- Premier corpus électronique anglais de plus d'un million de mots (1961, Brown University)
- **Contenu :** 500 sources catégorisées par genre (15) :
  - news : article d'information de journal
  - editorial : éditorial de journal
  - government : texte législatif
  - science_fiction : roman ou nouvelle de science-fiction
  - romance : roman ou nouvelle d'amour
  - humor : texte humoristique
  - ...
- **Utilité :** comparaison de styles linguistiques

Quelques corpus (2/2) : Gutenberg et Webtext

- Gutenberg : `nltk.corpus.gutenberg`
  - **Contenu** : 25.000 livres électroniques
  - **Utilité :** analyse de la langue écrite
- Webtext : `nltk.corpus.webtext`
  - **Contenu :** aggrégation de forums de discussion, conversations de la rue, scripts de film, annonces personnelles, et critiques
  - **Utilité :** étude de la langue non formelle (*i.e.* comment parlent "réellement" les gens)

Opérations simples sur un corpus composite

- `corpus` : un corpus composite déjà chargé
- `corpus.fileids()` : la liste des noms de fichiers contenus dans ce corpus
- `corpus.raw(fileid)` : le fichier `fileid` du `corpus` sous forme de chaîne de caractères
- `corpus.words(fileid)` : le fichier `fileid` du `corpus` sous forme de liste de mots
- `corpus.sents(fileid)` : le fichier `fileid` du `corpus` sous forme de liste de phrases, composées de listes de mots

Opérations simples sur un corpus composite

Exemple : fichiers contenus et différentes représentations

```
>>> nltk.corpus.gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt',
'austen-sense.txt', 'bible-kjv.txt', 'blake-poems.txt',
..., 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt',
'whitman-leaves.txt']
```

Analyse fréquentielle conditionnelle

Exemple : décompte des modaux dans les textes d'information

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

Analyse fréquentielle conditionnelle

Exemple : décompte des modaux dans différentes catégories

```
>>> cfd = nltk.ConditionalFreqDist(
...     (g, w)
...     for g in brown.categories()
...     for w in brown.words(categories=g))
>>> genres = ['news', 'religion', 'hobbies']
>>> modals = ['can', 'could', 'may', 'might', 'must',
'will']
>>> 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
```

Chargement de son propre corpus

Pour une tâche donnée, il faut un corpus adapté

```
>>> from nltk.corpus import PlaintextCorpusReader
>>> root = ' /monCorpus/'
>>> monCorpus = PlaintextCorpusReader(root, '.*')
>>> monCorpus.fileids()
'texte1', 'texte2', 'texte3'
```

Plan du cours

Introduction

Vocabulaire :

Étiquetage morpho-syntaxique = Part-of-Speech Tagging
= POS Tagging = Tagging

Objectifs :

- Étudier la deuxième étape d'une analyse de la langue naturelle classique après la tokenization
- Voir l'utilité concrète des catégories lexicales
- Déterminer comment stocker les mots et leurs catégories lexicales en Python
- Automatiser le processus d'étiquetage morpho-syntaxique

Utilisation d'un tagger

Un étiquetteur morpho-syntaxique attache un tag à chaque mot.

Découpage d'une chaîne de caractères en une liste de mots :

```
>>> text = "Hello to you, bright and shiny world!"
>>> tokens = nltk.word_tokenize(text)
>>> tokens
['Hello', 'to', 'you', ',', 'bright', 'and', 'shiny',
'world', '!']
```

Étiquetage d'une liste de mots en liste de tuples :

```
>>> nltk.pos_tag(tokens)
[('Hello', 'NNP'), ('to', 'TO'), ('you', 'PRP'), (',',
','), ('bright', 'JJ'), ('and', 'CC'), ('shiny', 'NN'),
('world', 'NN'), ('!', '.')  ]
```

Liste des 36 tags de NLTK (Penn Treebank)

Tags	Description	Tags	Description
CC	Coordinating conjunction	PRP$	Possessive pronoun
CD	Cardinal number	RB	Adverb
DT	Determiner	RBR	Adverb comparative
EX	Existential *there*	RBS	Adverb superlative
FW	Foreign word	RP	Particle
IN	Preposition / sub. conjunct.	SYM	Symbol
JJ	Adjective	TO	*to*
JJR	Adjective, comparative	UH	Interjection
JJS	Adjective, superlative	VB	Verb, base form
LS	List item marker	VBD	Verb, past tense
MD	Modal	VBG	Verb, gerund or pres. participle
NN	Noun, singular or mass	VBN	Verb, past participle
NNS	Noun, plural	VBP	Verb, non-3rd person sing. pres.
NNP	Proper noun, singular	VBZ	Verb, 3rd person sing. pres.
NNPS	Proper noun, plural	WDT	Wh-determiner
PDT	Predeterminer	WP	Wh-pronoun
POS	Possesive ending	WP$	Possessive wh-pronoun
PRP	Personal pronoun	WRB	Wh-adverb

```
>>> nltk.help.upenn_tagset(tag) # info sur le tag
```

Quelques remarques sur les tags

- Un même mot peut avoir plusieurs tags :
  "I like to ski(VB) races(NNS)" *vs*
  "I like to race(VB) on ski(NN)"
- Des mots d'une même catégorie grammaticale occurrent souvent dans des contextes similaires :

```
>>> text.similar('woman')
man time day year car moment world family house country child boy...
>>> text.similar('the')
a his this their its her an that our any all one these my...
```

Et en français ?

- NLTK n'inclut pas de tagger français, par défaut
- possibilité d'utiliser TreeTagger avec un wrapper Python. Voir : `http://www.fabienpoulard.info/post/2011/01/09/Python-et-Tree-Tagger`
- possibilité d'utiliser le Stanford POS tagger (à télécharger séparément depuis `https://nlp.stanford.edu/software/tagger.shtml#Download`)

Et en français ?

Utilisation de Stanford POS tagger en français

```
>>> root_stanfordpostagger = "chemin"
>>> root_jdk = "chemin"
>>> jar = root_stanfordpostagger +
"stanford-postagger-3.8.0.jar"
>>> model = root_stanfordpostagger +
"models/french.tagger"
>>> import os
>>> os.environ['JAVAHOME'] = root_jdk +
"jre/bin/java.exe" # Java 8 JDK requis
>>> pos_tagger = StanfordPOSTagger(model, jar,
encoding='utf8')
>>> res = pos_tagger.tag('bonjour le monde'.split())
>>> print(res)
[('bonjour', 'NC'), ('le', 'DET'), ('monde', 'NC')]
```

Corpus pré-taggés

La plupart des corpus fournis avec NLTK ont déjà été étiquetés :

```
>>> corpus.tagged_words() # liste de mots
>>> corpus.tagged_sents() # liste de phrases
```

Mais plusieurs ensembles d'étiquettes existent ! Pour se ramener à un ensemble commun :

```
>>> simplifiedTags = [(word, nltk.map_tag('en-ptb',
'universal', tag)) for word, tag in posTagged]
```

Ensemble de tags simplifiés (sous-ensemble de Penn Treebank) :
ADJ, ADV, CNJ, DET, EX, FW, MOD, N, NP, NUM, PRO, P, TO, UH, V, VD, VG, VN, WH

Exemple 1 : tags les plus fréquents

Quels sont les tags les plus fréquents dans la catégorie news du corpus Brown ?

```
>>> from nltk.corpus import brown
>>> brownNewsTagged =
brown.tagged_words(categories='news')
>>> tags_fd = nltk.FreqDist(tag for (word, tag) in
brownNewsTagged)
>>> tags_fd.keys()
['N', 'P', 'DET', 'NP', 'V', 'ADJ', ',', '.', 'CNJ',
'PRO', 'ADV', 'VD', ...]
```

Exemple 2 : quels tags co-occurrent avec les noms ?

Récupération des bigrammes du corpus :

```
>>> wordTagPairs = nltk.bigrams(brownNewsTagged)
>>> [p for p in wordTagPairs][11:13]
[(('recent', 'ADJ'), ('primary', 'NOUN')),
(('primary', 'NOUN'), ('election', 'NOUN'))]
```

Liste des tags précédant un nom :

```
>>> list(nltk.FreqDist(a[1] for (a, b) in wordTagPairs
if b[1] == 'NOUN'))
['DET', 'ADJ', 'N', 'P', 'NP', 'NUM', 'V', 'PRO', 'CNJ',
'.', ',', 'VG', 'VN', ...]
```

Liste des tags suivant un nom :

```
>>> list(nltk.FreqDist(b[1] for (a, b) in wordTagPairs
if a[1] == 'NOUN'))
['P', 'N', '.', ',', 'CNJ', 'V', 'VD', 'NP', 'ADV',
'MOD', "'", 'TO', 'DET, ...  ]
```

Exemple 3 : quels sont les verbes du corpus ?

Comptage de fréquence :

```
>>> brownNewsFd = nltk.FreqDist(brownNewsTagged)
>>> brownNewsFd
(FreqDist with 15551 samples and 100554 outcomes)
```

Affichage si le tag associé commence par 'V' :

```
>>> [word + "/" + tag for (word, tag) in brownNewsFd if
tag.startswith('V')]
['is/V', 'said/VD', 'was/VD', 'are/V', 'be/V', 'has/V',
'have/V', 'says/V', 'were/VD', 'had/VD', 'been/VN',
"'s/V", 'do/V', 'say/V', 'make/V', 'did/VD', ,...]
```

Exemple 4 : recherche de trigrammes particuliers

Quelles sont les séquences de la forme "V to V" dans le corpus de news Brown ?

```
  >>> for tagSent in brown.tagged_sents():  # pour toute
phrase
  >>>   for (w1,t1), (w2,t2), (w3,t3) in
nltk.trigrams(tagSent):  # pour tout trigramme
  >>>     if (t1.startswith('V') and t2=='TO' and
t3.startswith('V')):
  >>>       print(w1, w2, w3)
  ...
  trying to get
  going to jump
  refuses to continue
  ...
```

Un peu de Python : les dictionnaires

En Python, on a vu que les listes permettent de récupérer un élément par son indice :

```
>>> l = ['a', 'b', 'c']
>>> l[1]
'b'
```

Avec les dictionnaires, on peut récupérer une valeur par sa clé :

```
>>> d = {'a':2, 'b':4, 'c':6}
>>> d['b']
4
```

Il est possible de lister toutes les clés et valeurs :

```
>>> d.keys()
["a", "c", "b"]
>>> d.values()
[2, 6, 4]
```

Noter que le dictionnaire est, par nature, non ordonné.

Étiquetage automatique : objectifs

Le POS associé à un mot dépend :

1. du mot lui-même,
2. de son contexte (mots alentours).

Il faut donc travailler au niveau de la phrase.

Question : comment tagger un corpus ?

On continue de travailler sur les news du corpus Brown :

```
>>> from nltk.corpus import brown
>>> brownNewsTaggedSents =
brown.tagged_sents(categories='news')
>>> brownNewsSents = brown.sents(categories='news')
```

`brownNewsTaggedSents` servira de référence pour évaluer la performance de notre POS Tagger.

Approche #1 : Tagger naïf

Pour établir une baseline, nous allons tagger chaque mot avec le POS le plus fréquent dans le corpus :

```
>>> tags = [tag for (word, tag) in
brown.tagged_words(categories='news')]
>>> nltk.FreqDist(tags).max()
'NN'
```

Création d'un tagger qui étiquette tout mot comme un nom :

```
>>> naiveTagger = nltk.DefaultTagger('NN')
```

Application sur le corpus Brown de news :

```
>>> naiveTagger.evaluate(brownNewsTaggedSents)
0.13089484257215028
```

Test sur une phrase :

```
>>> tokens=nltk.word_tokenize("Hello world")
>>> naiveTagger.tag(tokens)
[('Hello', 'NN'), ('world', 'NN')]
```

Approche #2 : Tagger à base d'expressions régulières

On suppose que certaines terminaisons sont suffisantes pour déterminer le POS d'un mot.

```
>>> patterns = [
...   (r'.*ing$', 'VBG'), # gérondif
...   (r'.*ed$', 'VBD'), # passé simple
...   (r'.*es$', 'VBZ'), # 3e pers.  sing.  présent
...   (r'.*ould$', 'MD'), # modal
...   (r'.*´s$', 'NN$'), # nom possessif
...   (r'.*s$', 'NNS'), # nom pluriel
...   (r'^-?[0-9]+(.[0-9]+)?$', 'CD'), # nombre cardinal
...   (r'.*', 'NN') # nom (défaut)
...]
```

L'ordre des règles définit l'ordre d'application.

Approche #2 : Tagger à base d'expressions régulières

On suppose que certaines terminaisons sont suffisantes pour déterminer le POS d'un mot.

Évaluation de la performance :

```
>>> regExpTagger = nltk.RegexpTagger(patterns)
>>> regExpTagger.evaluate(brownNewsTaggedSents)
0.20326391789486245
```

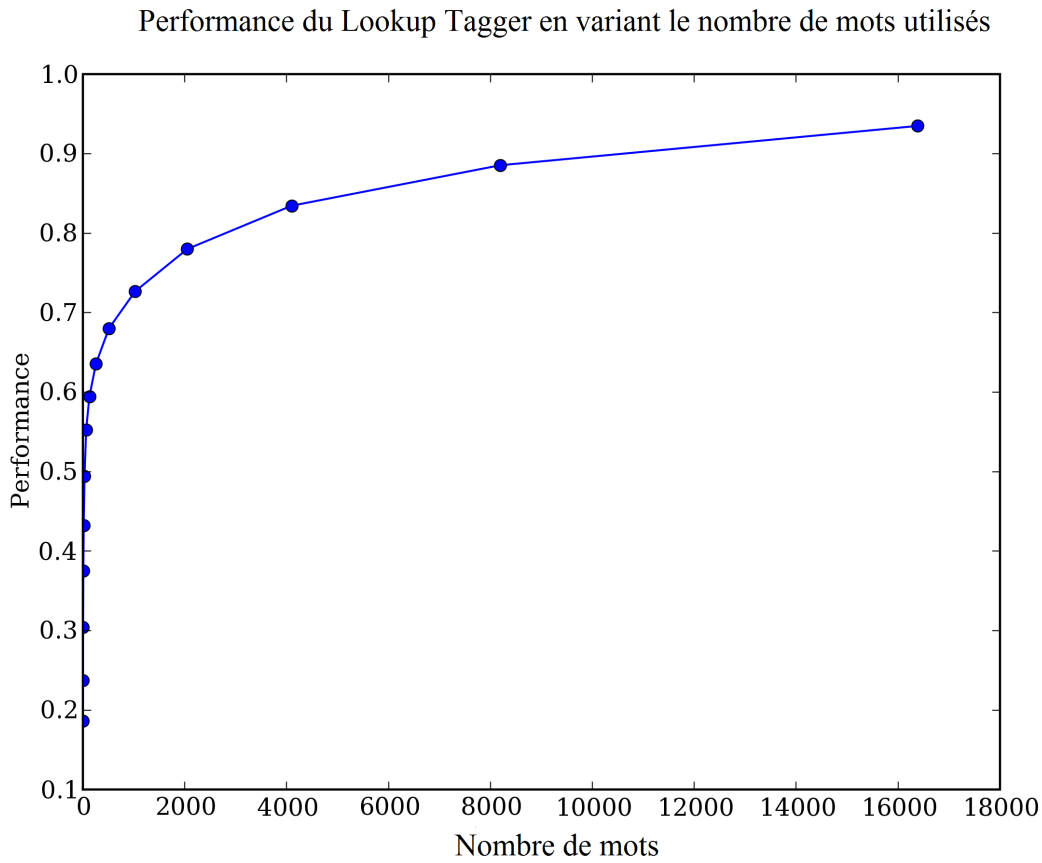
Approche #3 : Lookup tagger

On sait (loi de Zipf) que traiter correctement les 100 mots les plus courants permettrait une amélioration conséquente de la performance. On va donc tagger ceux-ci et leur associer leur tag le plus courant :

```
>>> fd = nltk.FreqDist(brown.words(categories='news'))
>>> cfd =
nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
>>> mostFreqWords = sorted(fd.items(),
key=operator.itemgetter(1))[::-1][:100]
>>> mostFreqWords = [k for (k,v) in mostFreqWords]
>>> likelyTags = dict((word, cfd[word].max()) for word
in mostFreqWords)
>>> baselineTagger =
nltk.UnigramTagger(model=likelyTags)
>>> baselineTagger.evaluate(brownNewsTaggedSents)
0.4557849513694134
```

Approche #3 : Lookup tagger

Procéder de même sur plus de 100 mots continuera d'améliorer les performances, mais de manière de moins en moins efficace.



Méta-approche : combinaison de taggers

Dans l'exemple précédent, seuls 100 mots sont étiquetés : il faudrait donc au moins étiqueter le reste avec le tagger naïf par défaut.

```
>>> baselineTaggerBO =
nltk.UnigramTagger(model=likelyTags,
backoff=nltk.DefaultTagger('NN'))
>>> baselineTaggerBO.evaluate(brownNewsTaggedSents)
0.5817769556656125
```

Approche #4 : Tagger à base de N-grammes

Il est possible de généraliser l'approche du lookup tagger en apprenant sur un sous-ensemble (9/10) du corpus et en testant sur le reste (1/10) :

```
>>> size = int(len(brownNewsTaggedSents) * 0.9)
>>> trainSents = brownNewsTaggedSents[:size]
>>> testSents = brownNewsTaggedSents[size:]
>>> unigramTagger = nltk.UnigramTagger(trainSents)
>>> unigramTagger.evaluate(testSents)
0.8120203329014252 8
```

Attention : apprendre sur tout le corpus donnerait un meilleur score, mais il serait erroné !

Approche #4 : Tagger à base de N-grammes

- On peut utiliser des bigrammes (`nltk.BigramTagger`) ou trigrammes (`nltk.TrigramTagger`) pour avoir plus d'information sur le contexte.
- La précision augmente avec N... tout comme la chance de ne pas avoir ces données dans l'ensemble d'apprentissage ! (problème des données creuses)

```
>>> bigramTagger = nltk.BigramTagger(trainSents)
>>> bigramTagger.evaluate(testSents)
0.1027608890660819 3
```

Mais on peut les combiner pour les cas non traités.

Gestion des mots inconnus

- Les approches à base de N-grammes ne peuvent pas gérer les mots inconnus $\longrightarrow$ besoin de combiner avec les 2 premières approches
- Un mot inconnu sera toujours identifié comme ayant la même étiquette $\longrightarrow$ besoin de tagger les mots peu courants en tant que UNK, pour permettre aux taggers à base de N-grammes de différencier les contextes (*e.g.* "to google" *vs* "the Google search engine")

Approche #5 : Tagger de Brill

- Le principal problème des N-grammes est la taille de la table en mémoire, et le fait qu'ils prennent en compte les mots individuels plutôt que les tags
- le tagger de Brill, à base de transformations successives, devine le tag de chaque mot (avec une approche unigramme) puis corrige les erreurs de plus en plus fines $\longrightarrow$ apprentissage supervisé
- les règles utilisées ont un sens linguistique, *e.g.* :
  - Replace NN with VB when the previous word is TO
  - Replace TO with IN when the next tag is NNS

Il est disponible via la commande `nltk.tag.brill`.

Plan du cours

Introduction

Vocabulaire :
Analyse syntaxique = analyse grammaticale = parsing

Objectifs :

- Étudier la troisième étape d'une analyse de la langue naturelle classique après le POS tagging
- Utiliser une grammaire pour décrire la structure d'un nombre illimité de phrases
- Représenter la structure d'une phrase sous forme d'arbre syntaxique
- Comprendre comment les analyseurs grammaticaux fonctionnent et construisent les arbres d'analyse syntaxique

Définition d'une grammaire hors contexte simple

Une grammaire simple est définie sous forme de règles, parsées par la commande `nltk.CFG.fromstring`).

Soit la phrase (ambiguë) : "I shot an elephant in my pajamas"
Elle nécessite une grammaire dans laquelle chaque mot est en partie droite d'une règle. S est le symbole de départ.

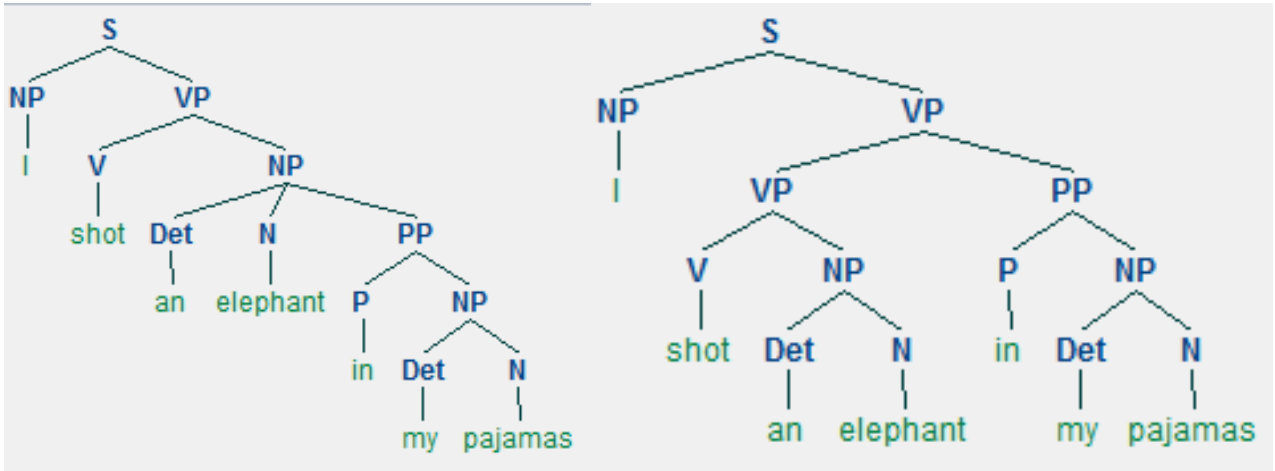
```
>>> gramm = nltk.CFG.fromstring("""
...   S -> NP VP
...   PP -> P NP
...   NP -> Det N | Det N PP | 'I'
...   VP -> V NP | VP PP
...   Det -> 'an' | 'my'
...   N -> 'elephant' | 'pajamas'
...   V -> 'shot'
...   P -> 'in'
...   """)
```

Utilisation d'une grammaire pour l'analyse

```
>>> sent = nltk.word_tokenize("I shot an elephant in my
pajamas")
>>> parser = nltk.ChartParser(gramm)
>>> trees = parser.parse(sent)
>>> for tree in trees:
...  print(tree)
(S
  (NP I)
  (VP
    (V shot)
    (NP (Det an) (N elephant) (PP (P in) (NP (Det my) (N
pajamas))))))
(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an) (N elephant)))
    (PP (P in) (NP (Det my) (N pajamas)))))
```

Utilisation d'une grammaire pour l'analyse (2)

```
>>> for tree in trees:
...     cf = nltk.draw.util.CanvasFrame()
...     t = nltk.Tree.fromstring(str(tree))
...     tc = nltk.draw.TreeWidget(cf.canvas(),t)
...     cf.add_widget(tc, 10, 10)
...     cf.print_to_file('arbre.ps')
```



Définition d'une grammaire dans un fichier externe

Pour faciliter l'écriture d'une grammaire personnalisée, il est préférable de l'enregistrer dans un fichier séparé `.cfg`, chargé par la commande `gramm = nltk.data.load('file:mygrammar.cfg')`

Quelques erreurs courantes :

- mélanger catégories grammaticales et éléments lexicaux en partie droite d'une règle (*e.g.* PP -> 'of' NP)
- utiliser des expressions de plus d'un mot (*e.g.*'New York' au lieu de 'New_York')

Grammaire récursive

Une grammaire est dite **récursive** si une catégorie présente en partie gauche d'une règle apparaît également en partie droite :

- récursion directe : Nom -> Adj Nom
- récursion indirecte : S -> NP VP et VP -> V S

Il n'y a pas de limite possible sur la profondeur de récursion.

Analyse d'une grammaire hors contexte : récursion descendante

Principes :

- découper un but de haut niveau (atteindre S) en plusieurs sous-buts de niveaux inférieurs,
- lorsqu'un mot est trouvé, essayer de l'associer à la phrase en entrée,
- quand une association mot/entrée n'est pas possible, on remonte dans la récursion (backtrack).

Démonstration dynamique : `nltk.app.rdparser()`

Pour utiliser le parseur à récursion descendante :

```
>>> rdParser = nltk.RecursiveDescentParser(gramm)
>>> for t in rdParser.nbest_parse(sent):
...   print t
```

Analyse d'une grammaire hors contexte : shift-reduce

Principes :

- rechercher des mots ou phrases qui matchent la partie droite des règles, et la remplace par la partie gauche,
- les mots sont déplacés (shift) un à un en haut d'une pile,
- à chaque étape, on regarde si une règle permet de réduire la pile (sur le dessus),
- arrêt quand l'entrée est vide ou la pile a atteint le symbole de départ.

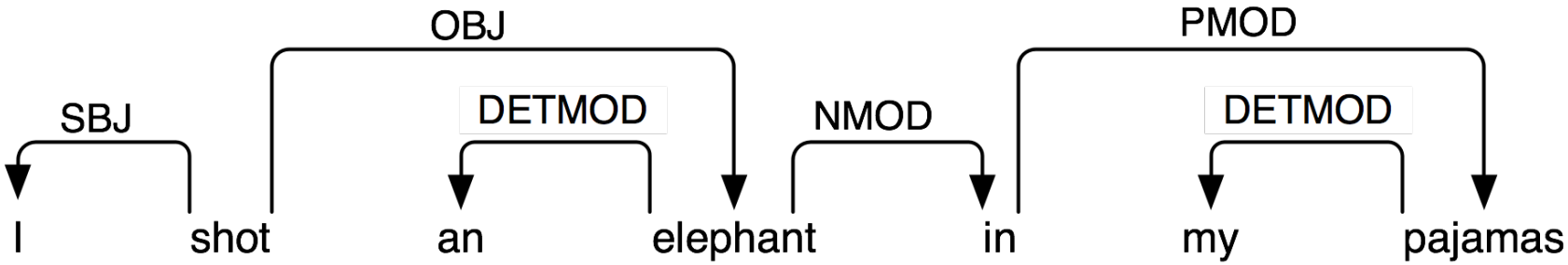
Démonstration dynamique : `nltk.app.srparser()`

Pour utiliser le parseur shift-reduce :

```
>>> srParser = nltk.ShiftReduceParser(gramm)
>>> print srParser.parse(sent)
```

Principes d'une grammaire de dépendance

- Une grammaire structurelle s'intéresse à la façon de combiner les mots
- Une grammaire de dépendance s'intéresse aux relations entre les mots
- La dépendance est assymétrique entre la tête (head - généralement le verbe) et ses dépendances
- La dépendance est représentée par des arcs étiquetés par une fonction grammaticale.
- Un graphe de dépendance est **projectif** si quand les mots sont écrits dans l'ordre, les arcs peuvent être dessinés au-dessus des mots sans croisement.



Définition d'une grammaire de dépendance simple

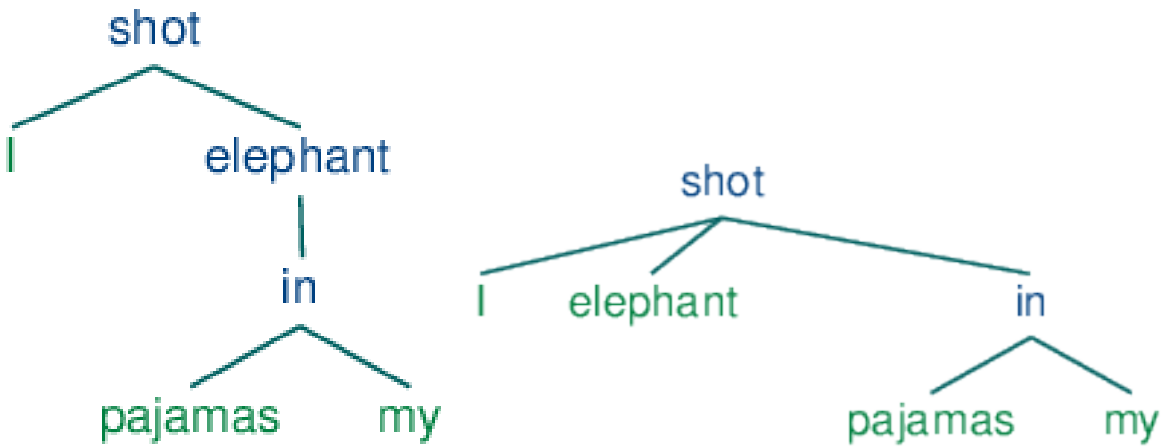
```
>>> depgramm = nltk.parse_dependency_grammar("""
... 'shot' -> 'I' | 'elephant' | 'in'
... 'elephant' -> 'an' | 'in'
... 'in' -> 'pajamas'
... 'pajamas' -> 'my'
... """)
```

Affichage des règles encodées :

```
>>> print depgramm
Dependency grammar with 7 productions
'shot' -> 'I'
'shot' -> 'elephant'
'shot' -> 'in'
'elephant' -> 'an'
'elephant' -> 'in'
'in' -> 'pajamas'
'pajamas' -> 'my'
```

Utilisation d'une grammaire de dépendance simple

```
>>> pdp = nltk.ProjectiveDependencyParser(gramm)
>>> trees = pdp.parse('I shot an elephant in my
pajamas'.split())
>>> for tree in trees:
. . .   print tree
(shot I (elephant an (in (pajamas my))))
(shot I (elephant an) (in (pajamas my)))
```



Valences et lexique

Soient les phrases suivantes :

1. The cat was asleep.
2. John saw the cat.
3. John thought Mary was angry.
4. John put the cat on the mat.

Les règles de production suivantes seraient nécessaires à la grammaire :

1. VP $\rightarrow$ V Adj #was
2. VP $\rightarrow$ V NP #saw
3. VP $\rightarrow$ V S #thought
4. VP $\rightarrow$ V NP PP #put

Les dépendances Adj, NP, S, et PP sont appelés les compléments du verbe. On dit que les verbes ont des **valences** différentes.

Valences et lexique (2)

Pour gérer les contraintes liées à la valence, dans une CFG, on aura recours à des catégories différentes de verbes. Par exemple :

- les verbes transitifs (TV) :
  - VP -> TV NP
  - TV -> 'saw'
- les verbes intransitifs (IV) : to bark
- les verbes datifs (DatV) : to give something to someone
- les verbes sententiels (SV) : to say that something

Passage à l'échelle des grammaires ?

Problème : comment complexifier des grammaires pour gérer plus de cas ?

- On peut définir des grammaires couvrant des micro-mondes.
- Le passage à l'échelle est très difficile, à cause des interactions complexes entre règles : les grammaires ne sont **pas modulaires**.
- Plus il y a de règles, plus le nombre d'interprétations augmente : **l'ambiguïté augmente avec la couverture**.

  Quelques projets collaboratifs ont tout de même obtenus de bons résultats :

  - Lexical Functional Grammar (LFG) : Projet Pargram
  - Head-Driven Structure Grammar (HPSG) : LinGO Matrix
  - Lexicalized Tree Adjoining Grammar (LTAG) : Projet XTAG

Treebanks

Problème : comment appliquer des grammaires à un large ensemble de phrases ?

On appelle **Treebank** un corpus annoté grammaticalement ou sémantiquement (*e.g.* le Penn Treebank corpus, partiellement inclus dans NLTK).

```
>>> from nltk.corpus import treebank
>>> t = treebank.parsed_sents()    # t est une liste des phrases analysées
```

Treebanks (2)

Le Prepositional Phrase Attachment Corpus (`nltk.corpus.ppattach`) fournit des informations sur les valences des verbes.

Exemple : Recherche du rattachement au verbe ou au nom du PP des trigrammes N-PP-N

```
>>> entries = nltk.corpus.ppattach.attachments('training')
>>> table = nltk.defaultdict(lambda: nltk.defaultdict(set))
>>> for entry in entries:
...     key = entry.noun1 + '-' + entry.prep + '-' + entry.noun2
...     table[key][entry.attachment].add(entry.verb)
```

Treebanks (2)

Le Prepositional Phrase Attachment Corpus (`nltk.corpus.ppattach`) fournit des informations sur les valences des verbes.

Exemple : Recherche du rattachement au verbe ou au nom du PP des trigrammes N-PP-N

Affichage :

```
>>> for key in sorted(table):
...   if len(table[key]) > 1:
...     print key, 'N:', sorted(table[key]['N']), 'V:',
sorted(table[key]['V'])
```

Interprétation :

```
offer-from-group N: ['rejected'] V: ['received']
```

signifie qu'avec "to reject", from se rattache à offer, tandis qu'avec "to receive", il se rattache au verbe lui-même.