TELECOM Bretagne

F3B403A + Masters CSDS et IABDA

TP de fouille de texte

Yannis Haralambous (Télécom Bretagne)

Dans ce TP on se servira du langage Python et du (très riche) module NLTK (*Natural Language Toolkit*).

La documentation de NLTK se trouve là : `http://nltk.org/api/nltk.html`, ne pas hésiter d'utiliser la fonction de recherche.

La documentation de Python se trouve là : `http://docs.python.org/2/index.html`.

Pour installer NLTK sans être root, on écrira

```
pip install nltk --user
```

Puis, pour charger les corpus dont on aura besoin, on lancera Python en mode interactif, et on écrira :

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

Dans la fenêtre qui va s'ouvrir on choisira «all» dans «Collections» si on n'a pas de problème de quota, ou alors «book», et on cliquera sur «Donwload». Une fois l'opération terminée, pour quitter le mode interactif on tapera Ctrl-D.

1 Quelques aspects du corpus de Brown

On utilisera les modules suivants :

```
import nltk
from nltk.corpus import brown
from nltk.probability import FreqDist
import re
```

Les tags du corpus de Brown se trouvent là :
`http://www.comp.leeds.ac.uk/amalgam/tagsets/brown.html`.

Questions :

a) Dans le corpus de Brown, quels noms (*noun*) sont plus communs au pluriel qu'au singulier ? (On considère que le pluriel est formé simplement par l'ajout d'un «s») Classer les 20 premier résultats par fréquence de la forme plurielle et par ratio pluriel/singulier.

b) Quel mot a le plus grand nombre de tags distincts ? Que représentent-ils ?

c) Énumérer les tags par ordre de fréquence décroissante (les 20 premiers). Que représentent-ils ?

d) Quels sont les tags les plus fréquents de mots précédant des noms ? Que représentent-ils ?

2 Évaluation de taggeurs

2.1 Taggeurs de POS basés sur des n-grammes

Dans cette section on va créer des taggeurs (le taggeur par défaut est `nltk.DefaultTagger` et celui à n-grammes [1], `nltk.NgramTagger`), et on va les entraîner en utilisant la catégorie `news` du corpus de Brown.

Évaluer des taggeurs à n-grammes (pour $n = 0, \ldots, 6$) en effectuant une validation croisée décuple, avec et sans backoff.

Évaluer les mêmes taggeurs avec l'ensemble de tags simplifié.

2.2 Taggeurs de genre de prénom basés sur des propriétés intrinsèques

Dans cette sections on va utiliser les modules

```
import nltk
from nltk.corpus import names
import random
import collections
```

Le corpus `names` consiste en deux fichiers, `female.txt` et `male.txt`. Créer une liste de tuples (prénom,genre) dans un ordre aléatoire. Compter les prénoms mixtes épicènes.

Dans la section précédente on s'est servi de mots et de tags. Ici on adoptera une approche plus générale : on va définir des *propriétés* et on va évaluer l'apport de ces propriétés à la tâche de classification (dans notre cas : le genre d'un prénom). Première propriété candidate : la dernière lettre.

```
def gender_features(word):
   return {'last_letter': word[-1]}
featuresets = [(gender_features(n), g) for (n,g) in names]
train_set, test_set = featuresets[500:], featuresets[:500]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print nltk.classify.accuracy(classifier, test_set)
```

Ici on obtient l'exactitude. Calculer également la précision, le rappel et la F-mésure pour chaque genre (utiliser les méthodes homonymes de `nltk.metrics`, attention : pour utiliser ces méthodes il faut rassembler les numéros des prénoms dans des listes, selon leurs classements et leurs véritables genres). Que constate-t-on ?

À l'aide de la méthode `show_most_informative_features(10)` de `classifier`, on peut savoir quelles sont les valeurs des propriétés qui ont le plus de pouvoir discriminant.

Pour avoir des idées de nouvelles propriétés à tempter, écrire le code qui va afficher toutes les erreurs de classif avec, pour chacune, la classe correcte, la classe mal sélectionnée, le nom.

Tester d'autres propriétés, ainsi que leurs combinaisons.

2.3 Retour aux POS, approche combinée (propriétés intrinsèques + n-grammes)

Dans la section 2.1 nous avons créé des taggeurs entraînés sur les mots ainsi que sur un certain nombre de tags de mots précédents. Dans la section 2.2 nous avons (pour une autre tâche de classification : il ne s'agissait pas de classer par POS mais par seulement par genre) utilisé des propriétés intrinsèques des mots (décrites par des expressions régulières). Dans cette section nous allons combiner les deux approches pour le calcul du POS.

Commençons par créer un classificateur de POS basé uniquement sur des propriétés intrinsèques des mots : les 1-, 2- et 3-suffixes. Quelle exactitude obtient-on ?

Ensuite, ajoutons d'autres propriétés :

1. le mot courant,

1. Un taggeur à n-grammes est un classificateur qui utilise comme propriétés le mot courant et les tags des $n-1$ mots précédents. Un taggeur par défaut est un classificateur trivial qui affecte le même tag à tous les mots, on fait en sorte que ce soit le tag le plus fréquent du corpus.

2. le mot précédent,
3. le mot courant et le mot précédent,
4. les 1–3 ainsi que le tag du mot précédent, tel qu'il a été prédit par le classificateur,
5. idem mais avec les tags des deux mots précédents (en tant que propriétés séparées).

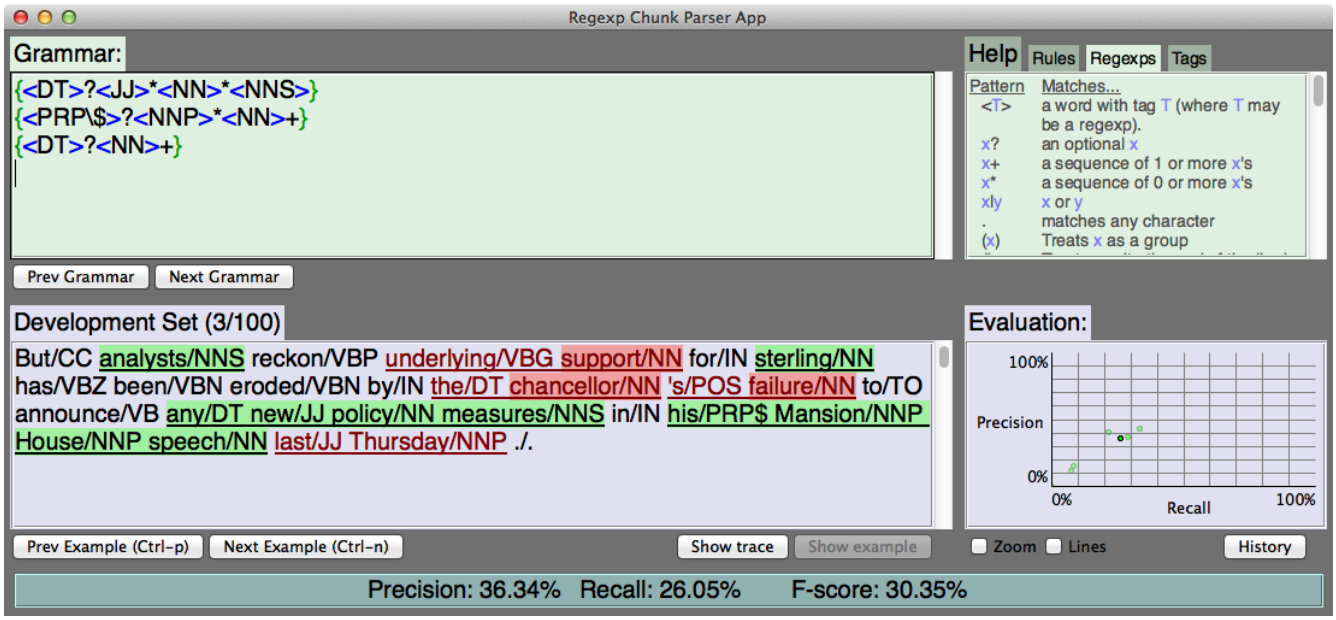
Comment les performances du classificateur évoluent-elles ?

Pour stocker les tags prédits, on peut utiliser une liste `history` (au niveau de chaque phrase).

3 Chunking, chinking

Dans la suite nous allons attaquer le *chunking* en écrivant des grammaires. Cela nous servira à une branche important du text mining, l'extraction d'information. Nous allons obtenir des chunks pronominaux et verbaux, et en cherchant des motifs de chunks on fera de l'extraction de relations.

Tout d'abord, servons-nous de l'interface *Regexp Chunk Parser App* pour nous entraîner à écrire des expressions régulières pour décrire des chunks. L'application nous donne en temps réel, la précision, le rappel et la F-mésure obtenus.

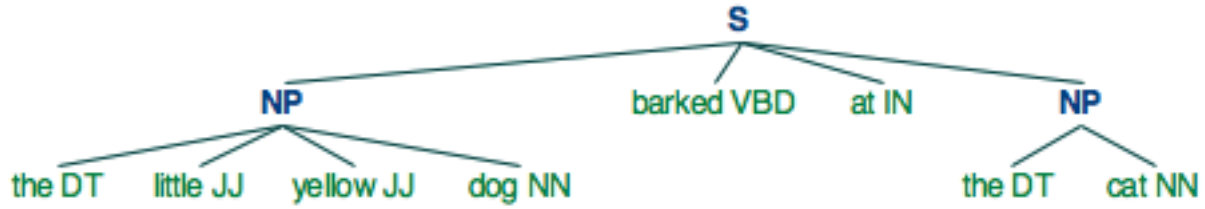


On la fait apparaître en écrivant

```
python
>>> import nltk
>>> nltk.app.chunkparser()
```

Voilà comment appliquer une telle grammaire, avec, le cas échéant, des labels de chunk, à une petite phrase *the/DT little/JJ yellow/JJ dog/NN barked/VBD at/IN the/DT cat/NN*. On instancie un parseur en lui fournissant une grammaire : il va donc lire les tags des mots et former des chunks selon la grammaire.

```
import nltk, re
sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),
("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
grammar = r"NP: {<DT>?<JJ>*<NN>}"
cp = nltk.RegexpParser(grammar)
result = cp.parse(sentence)
print result
result.draw()
```



Étendre la grammaire ci-dessus pour obtenir

à partir de la phrase *Rapunzel/NNP let/VBD down/RP her/PP$ long/JJ golden/JJ hair/NN*.

3.1 Évaluation de chunkeurs

Le corpus `conll2000` contient des phrases chunkées extraites du *Wall Street Journal*. Il est déjà divisé en un ensemble d'entraînement et un ensemble de test (fichiers `train.txt` et `test.txt`).

On va commencer par un chunkeur de NP basé sur une grammaire comme dans la section précédente :

```
from nltk.corpus import conll2000
cp = nltk.RegexpParser("")
test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
print cp.evaluate(test_sents)
```

L'exécuter. Comment expliquer le résultat obtenu ? Écrire quelques règles pour améliorer le résultat.

Une autre solution consiste à entraîner un chunkeur. On peut par exemple définir

```
class UnigramChunker(nltk.ChunkParserI):
   def __init__(self, train_sents):
      train_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(sent)]
      for sent in train_sents]
      self.tagger = nltk.UnigramTagger(train_data)
   def parse(self, sentence):
      pos_tags = [pos for (word,pos) in sentence]
      tagged_pos_tags = self.tagger.tag(pos_tags)
      chunktags = [chunktag for (pos, chunktag) in tagged_pos_tags]
      conlltags = [(word, pos, chunktag) for ((word,pos),chunktag)
      in zip(sentence, chunktags)]
      return nltk.chunk.util.conlltags2tree(conlltags)
```

Ici, `conlltags2tree` (et son inverse `tree2conlltags`) servent à passer du format IOB de ConLL2000 au format arborescent et vice-versa.

Est-ce que ce chunkeur fait intervenir les mots ? Pourquoi est-ce un taggeur d'«unigrammes» ? Évaluez-le, en utilisant les fichiers `train.txt` et `test.txt`.

Écrire des taggeurs de bigrammes et de trigrammes (sans backoff, hélas...) et comparer les résultats.