IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

UE TLFT

Lab 8: Frequentist Methods
(Wednesday March 10th, 2021)

Yannis Haralambous (IMT Atlantique)

To install NLTK without being root, you will write

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

Then, to load the corpora we need, we will launch Python in interactive mode, and write:

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

In the window that will open, choose "All" in "Collections" if you don't have quota problems, or "Book," and click on "Donwload". Once you have finished the operation, to exit the interactive mode, press Ctrl-D.

1 Some aspects of Brown's corpus

The following modules will be used:

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

Brown's corpus tags are here : `http://www.comp.leeds.ac.uk/amalgam/tagsets/brown.html`.

Questions and Answers:

a) In Brown's corpus, what nouns are more common in the plural than in the singular? (The plural is considered to be formed simply by the addition of an "s") Classify the first 20 results by frequency of the plural form and by plural/singular ratio.

b) Which word has the greatest number of distinct tags? What do they represent?

c) List the tags in descending order of frequency (the first 20). What do they represent?

d) What are the most frequent tags of words preceding nouns? What do they represent?

2 Evaluation of taggers

2.1 POS-based POS tags

In this section we will create taggers (the default tagger is `nltk.DefaultTagger` and the n-grams tagger[1], `nltk.NgramTagger`), and we will train them using the `news` category from Brown's corpus.

Evaluate taggers at n-grams (for $n = 0, \ldots, 6$) by performing a tenfold cross-validation, with and without backoff.

Evaluate the same taggers with the simplified tag set.

2.2 Given name gender tags based on intrinsic properties

In this section we will use modules

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

The `names` corpus consists of two files, `female.txt` and `male.txt`. Create a list of tuples (given name, gender) in a random order. Count the epicene mixed first names.

In the previous section we used words and tags. Here we will adopt a more general approach: we will define *properties* and we will evaluate the contribution of these properties to the classification task (in our case: the gender of a given name). First candidate property: the last letter.

```
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)
```

What we get is accuracy. Calculate also the precision, recall and F-measure for each gender (use the homonymous methods of `nltk.metrics`, beware: to use these methods you have to gather the numbers of given names in lists, according to their rankings and their true gender). What do we find?

Using the method `show_most_informative_features(10)` of `classify`, we can find out the values of properties having the most discriminating power.

To have ideas of new properties to attempt, write the code that will display all the classification errors with, for each one, the correct class, the badly selected class and the name.

Test other properties and their combinations.

2.3 Return to POS, combined approach (intrinsic properties + n-grams)

In Section 2.1 we have created taggers trained on words as well as on a number of previous word tags. In Section 2.2 we have (for another classification task: it was not a matter of classifying by POS but only by gender) used intrinsic properties of words (described by regular expressions). In this section we will combine the two approaches for POS computation.

Let us start by creating a POS classifier based only on intrinsic properties of words: 1-, 2- and 3-suffixes. What accuracy do we get?

Next, let us add more properties :

1. the current word,

2. the previous word,

[1] An n-gram tagger is a classifier that uses as properties the current word and the tags of the previous $n-1$ words. A default tagger is a trivial classifier that assigns the same tag to all words, we make sure that it is the most frequent tag in the corpus.

3. the current word and the previous word,

4. 1–3 as well as the tag of the previous word, as predicted by the classifier,

5. idem but with the tags of the two previous words (as separate properties).

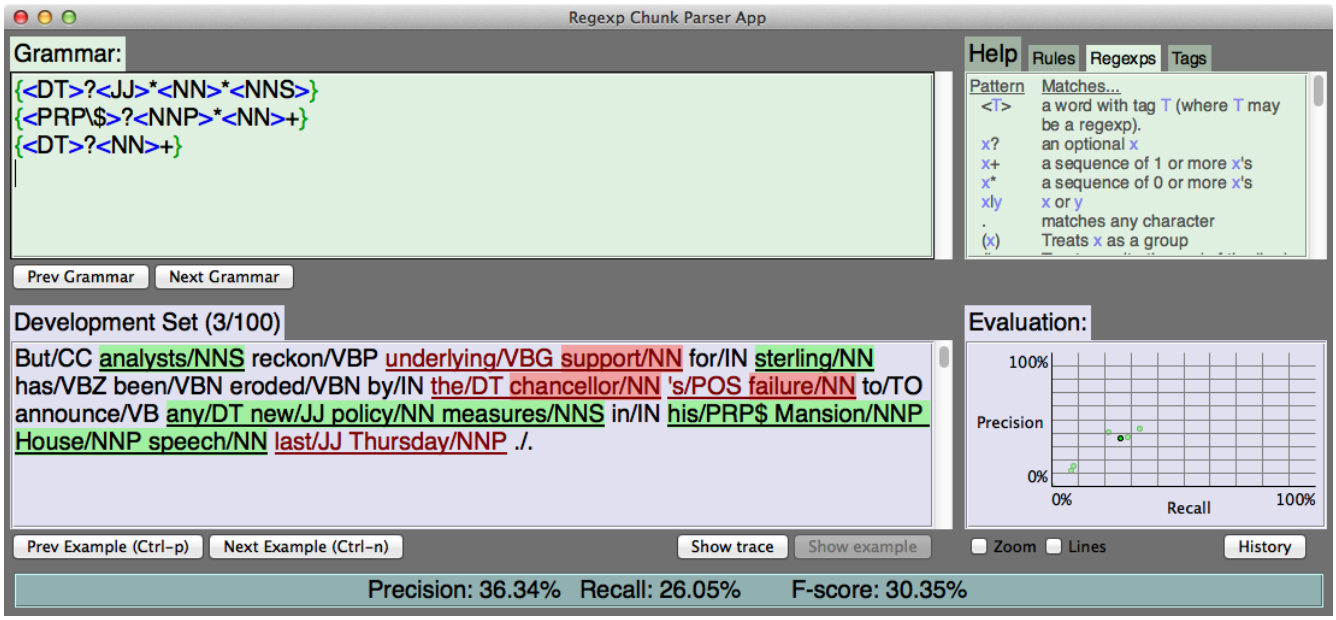
How does the performance of the classifier evolve?

To store the predicted tags, we can use a `history` list (at the level of each sentence).

3 Chunking, chinking

In the following we will do *chunking* by writing grammars. This will help us with an important branch of text mining, information extraction. We are going to get pronominal and verbal chunks, and by looking for patterns of chunks we will do relation extraction.

First of all, let us use the *Regexp Chunk Parser App* interface to practice writing regular expressions in order to describe chunks. The application gives us in real time, the values of accuracy, recall and F-measure obtained.

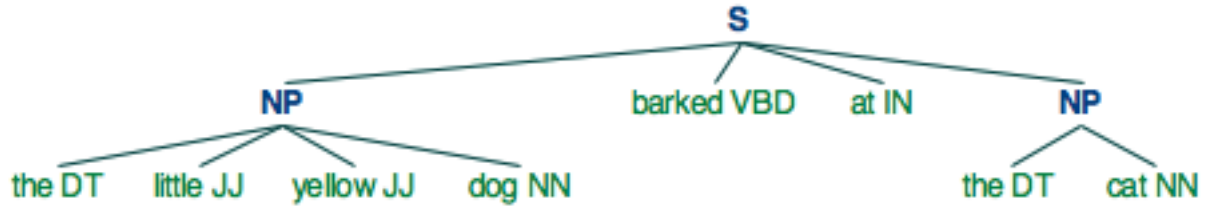


It gets displayed when we type

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

Here is how to apply such a grammar, with chunk labels if necessary, to a small sentence: *the/DT little/JJ yellow/JJ dog/NN barked/VBD at/IN the/DT cat/NN*. We instantiate a parser by providing it with a grammar: it will therefore read the word tags and form chunks according to the grammar.

```
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()
```



Extend the grammar to obtain

out of the sentence *Rapunzel/NNP let/VBD down/RP her/PP$ long/JJ golden/JJ hair/NN*.

3.1 Evaluation of chunkers

The corpus `conll2000` contains chunked sentences from the *Wall Street Journal*. It is already divided into a training set and a test set (files `train.txt` and `test.txt`).

We will start with an NP chunker based on a grammar as in the previous section:

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

Execute it. How do you explain the result obtained? Write some rules to improve the result.

Another solution is to train a chunker. One can, for example, define

```
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)
```

Here, `conlltags2tree` (and its inverse `tree2conlltags`) are used to switch from the IOB format of ConLL2000 to the tree format and vice versa.

Does this chunker involve words? Why is it a "unigram" tagger? Evaluate it, using the `train.txt` and `test.txt` files.

Write bigram and trigram taggers (without backoff, alas...) and compare the results.