NLTK Basic Tutorial

Introduction

This tutorial instruct you how to perform basic text processing with nltk (http://www.nltk.org) - a suite of libraries and programs for Natural Language Processing for English.

Download and Install nltk

You can install nltk by using `pip` (See http://www.nltk.org/install.html (http://www.nltk.org/install.html)). NLTK is included in Anaconda (https://www.continuum.io/downloads). I recommend you to use Anaconda to ease python package management tasks.

After downloading nltk, you should install nltk.data (http://www.nltk.org/data.html).

Basic Text Processing Tasks

Natural Language Content Analysis is a fundamental step in every text mining project. Depending on the text mining task, you may want to generate representations of text data in different levels. For instance, sentences in a text data may be simply represented as a **bag of words** or may be annotated with semantic word classes.

In this tutorial, we will learn how to use nltk to perform:

- Sentence segmentation
- Word Tokenization
- Word Lemmatization
- Word Stemming
- Filtering stop words.
- Part-of-speech tagging
- Extracting information from text
  - Chunking
  - Named Entity Recognition
  - Relation Extraction
- Analyzing structures of sentences
  - Phrasal structure analysis
  - Dependency parsing

Sentence segmentation

In many cases, we want to split a document or paragraph into a list of sentences. For instance, we want to identify sentiment of a sentence in the sentiment analysis task, or we may want to analyze structures of sentences.

To illustrate how to do it with `nltk`, we first create a paragraph.

In [1]:
```
para = "Hello World. It's good to see you. Thanks for buying this book."
```

Now, we want to split `para` into sentences. We will use module `nltk.tokenize` to do that.

In [2]:
```
from nltk.tokenize import sent_tokenize
sent_tokenize(para)
```

Out[2]:
```
['Hello World.', "It's good to see you.", 'Thanks for buying this book.']
```

Now, we have a list of sentences for further processing.

The instance used in `sent_tokenize()` is actually loaded on demand from a pickle file. So if you're going to be tokenizing a lot of sentences, it's more efficient to load the `PunktSentenceTokenizer` once, and call its `tokenize()` method instead.

In [3]:
```
import nltk.data
tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')
tokenizer.tokenize(para)
```

Out[3]:
```
['Hello World.', "It's good to see you.", 'Thanks for buying this book.']
```

Tokenizing a sentence into words

In this section, we will tokenize a sentence into words. We can do the task with the basic word tokenization by using the function `word_tokenization`.

In [4]:
```
from nltk.tokenize import word_tokenize
sent = 'The history of NLP generally starts in the 1950s, although work can be
found from earlier periods.'
print( word_tokenize(sent) )
```

```
['The', 'history', 'of', 'NLP', 'generally', 'starts', 'in', 'the', '1950s', '
,', 'although', 'work', 'can', 'be', 'found', 'from', 'earlier', 'periods', '.
']
```

We obtain a list of tokens as above.

`word_tokenize()` is a wrapper function that calls `tokenize()` on an instance of the `TreebankWordTokenizer`. It's equivalent to the following:

In [5]:
```
from nltk.tokenize import TreebankWordTokenizer
tokenizer = TreebankWordTokenizer()
print(tokenizer.tokenize(sent))
```

```
['The', 'history', 'of', 'NLP', 'generally', 'starts', 'in', 'the', '1950s', '
,', 'although', 'work', 'can', 'be', 'found', 'from', 'earlier', 'periods', '.
']
```

`TreebankWordTokenizer` use conventions in Penn Treebank corpus. We can have many alternatives for word tokenization.

In [6]:
```
word_tokenize("I can't swim.")
```

Out[6]:
```
['I', 'ca', "n't", 'swim', '.']
```

Alternatives of word tokenization

WordPunctTokenizer

`WordPunctTokenizer` splits all punctuations into separate tokens.

In [7]:
```
from nltk.tokenize import WordPunctTokenizer
tokenizer = WordPunctTokenizer()
tokenizer.tokenize("I can't swim.")
```

Out[7]:
```
['I', 'can', "'", 't', 'swim', '.']
```

RegexpTokenizer

We can use regular expression to tokenize words in a sentence.

In [8]:
```
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer("[\w']+")
tokenizer.tokenize("I can't swim.")
```

Out[8]:
```
['I', "can't", 'swim']
```

Or we can do in a simpler way.

In [9]:
```
from nltk.tokenize import regexp_tokenize
regexp_tokenize("I can't swim.", "[\w']+")
```

Out[9]:
```
['I', "can't", 'swim']
```

We can use simple whitespaces as delimiters for word tokenization.

In [10]:
```
tokenizer = RegexpTokenizer('\s+', gaps=True)
tokenizer.tokenize("I can't swim.")
```

Out[10]:
```
['I', "can't", 'swim.']
```

Word Lemmatization

As a definition, lemmatization is to "**remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma.**"

First, we create a raw text and tokenize it into tokens using the function `word_tokenize`.

In [11]:
```
raw = """DENNIS: Listen, strange women lying in ponds distributing swords ...
      is no basis for a system of government. Supreme executive power derives f
rom ...
      a mandate from the masses, not from some farcical aquatic ceremony."""
tokens = word_tokenize(raw)
```

We use WordNet lemmatizer for word lemmatization. The WordNet lemmatizer removes affixes only if the resulting word is in its dictionary.

In [12]:
```
wnl = nltk.WordNetLemmatizer()
print( [wnl.lemmatize(t) for t in tokens] )
```

```
['DENNIS', ':', 'Listen', ',', 'strange', 'woman', 'lying', 'in', 'pond', 'dis
tributing', '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', '.']
```

Stemming

TODO: Write up some motivations of the task.

Stemming is to chop off ends of words using some rules. In NLTK, we have several Stemmer to do the job. The following code will try two stemmers in nltk.

In [13]:
```
porter = nltk.PorterStemmer()
lancaster = nltk.LancasterStemmer()
print('++ Using PorterStemmer ++\n')
print([porter.stem(t) for t in tokens])
print('\n++ Using LancasterStemmer ++\n')
print([lancaster.stem(t) for t in tokens])
```

```
++ Using PorterStemmer ++

['DENNI', ':', 'Listen', ',', 'strang', 'women', 'lie', 'in', 'pond', 'distrib
ut', 'sword', '...', 'is', 'no', 'basi', 'for', 'a', 'system', 'of', 'govern',
 '.', 'Suprem', 'execut', 'power', 'deriv', 'from', '...', 'a', 'mandat', 'fro
m', 'the', 'mass', ',', 'not', 'from', 'some', 'farcic', 'aquat', 'ceremoni',
'.']

++ Using LancasterStemmer ++

['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', '.']
```

Filtering stop words

Stop words are common words that do not contribute to the meaning of a sentence. In general, earch engines and text mining systems filter stop words in the preprocessing step.

We can do that by creating a set of English stop words.

In [14]:
```
from nltk.corpus import stopwords
english_stops = set(stopwords.words('english'))
words = ["Can't", 'is', 'a', 'contraction']
[word for word in words if word not in english_stops]
```

Out[14]:
```
["Can't", 'contraction']
```

Part-of-speech tagging

Part-of-speech tagging is a process of assigning tags to words in a sentence. In this section, we will instruct you to do tasks as follows.

- Explore Tagged Corpora (Brown Corpus)
- Perform automatic tagging
- Train POS Tagger with (some) basic methods:
  - Ngram tagging (unigram, bigram, tagging)
  - Transformation-based tagging (Brill tagging)
  - Classifier based tagging

Explore Tagged Corpora (Brown Corpus)

Several corpora included with NLTK have been tagged for their part-of-speech. Brown Corpus is an example. If you open a file in the Brown Corpus, you can see tagged sentences like the following example.

```
The/at Fulton/np-tl County/nn-tl Grand/jj-tl Jury/nn-tl said/vbd Friday/nr an/at
investigation/nn of/in Atlanta's/np$ recent/jj primary/nn election/nn produced/vbd ``/``
no/at evidence/nn ''/'' that/cs any/dti irregularities/nns took/vbd place/nn ./.
```

Each token in the sentence is associated with a POS tag.

Now we show some tagged words in the corpus.

```
In [15]: import nltk
         nltk.corpus.brown.tagged_words()
```

```
Out[15]: [('The', 'AT'), ('Fulton', 'NP-TL'), ...]
```

Show tagged words with universal tagset.

```
In [16]: nltk.corpus.brown.tagged_words(tagset='universal')
```

```
Out[16]: [('The', 'DET'), ('Fulton', 'NOUN'), ...]
```

For further exploration of Brown corpus see the section 2.8, chapter 5 of the book *Natural Language Processing with Python* (Link: http://www.nltk.org/book/ch05.html (http://www.nltk.org/book/ch05.html))

Perform automatic tagging

In `nltk`, we can perform automatic POS tagging as follows.

```
In [17]: import nltk
         text = word_tokenize("And now for something completely different")
         print( nltk.pos_tag(text) )
```

```
         [('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completel
         y', 'RB'), ('different', 'JJ')]
```

The default pos tagger model using in NLTK is maxent_treebank_pos_tagger model (Maxent model trained on the treebank corpus).

Training POS taggers

In this section, we will train POS taggers using Brown corpus. Because of the computational time reason, we select sentences in the category "news". You can try to use the whole Brown Corpus.

An important thing in NLP projects is evaluating the performance of the trained model on a gold standard test set. Thus, in the first step, we seperate the corpus into training and test set.

In [18]:
```
from nltk.corpus import brown
brown_tagged_sents = brown.tagged_sents(categories='news')
brown_sents = brown.sents(categories='news')
# You can try to use the whole Brown corpus
# brown_tagged_sents = brown.tagged_sents()
# brown_sents = brown.sents()

size = int(len(brown_tagged_sents) * 0.9)
train_sents = brown_tagged_sents[:size]
test_sents = brown_tagged_sents[size:]

print('Training set size = %d' % size)
print('Test set size = %d' % (len(brown_sents) - size) )
```

```
Training set size = 4160
Test set size = 463
```

Unigram POS Tagging

Unigram taggers are based on a simple statistical algorithm: for each token, assign the tag that is most likely for that particular token. For example, it will assign the tag JJ to any occurrence of the word frequent, since frequent is used as an adjective (e.g. a frequent word) more often than it is used as a verb (e.g. I frequent this cafe).

We train an unigram tagging model and evaluate the trained model on the test set using the following code.

In [19]:
```
unigram_tagger = nltk.UnigramTagger(train_sents)
unigram_tagger.evaluate(test_sents)
```

Out[19]:
```
0.8129173726701884
```

Bigram POS tagging

In the general n-gram taggers, we use the context that contain the current token along with part-of-speech tags of the n-1 preceding tokens.

In [20]:
```
bigram_tagger = nltk.BigramTagger(train_sents)
print(bigram_tagger.tag(brown_sents[2007]))
```

```
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ('apartments', 'NNS'), ('are'
, 'BER'), ('of', 'IN'), ('the', 'AT'), ('terrace', 'NN'), ('type', 'NN'), (','
, ','), ('being', 'BEG'), ('on', 'IN'), ('the', 'AT'), ('ground', 'NN'), ('flo
or', 'NN'), ('so', 'CS'), ('that', 'CS'), ('entrance', 'NN'), ('is', 'BEZ'), (
'direct', 'JJ'), ('.', '.')]
```

Using the trained bigram tagging model to tag unseen data.

In [21]:
```
unseen_sent = brown_sents[4203]
print( bigram_tagger.tag(unseen_sent) )
```

```
[('The', 'AT'), ('population', 'NN'), ('of', 'IN'), ('the', 'AT'), ('Congo', '
NP'), ('is', 'BEZ'), ('13.5', None), ('million', None), (',', None), ('divided
', None), ('into', None), ('at', None), ('least', None), ('seven', None), ('ma
jor', None), ('``', None), ('culture', None), ('clusters', None), ("''", None)
, ('and', None), ('innumerable', None), ('tribes', None), ('speaking', None),
('400', None), ('separate', None), ('dialects', None), ('.', None)]
```

We can see that the bigram tagger is quite percise, but its coverage is low. Thus, the overall accuracy on the test data in very low.

In [22]:
```
bigram_tagger.evaluate(test_sents)
```

Out[22]:
```
0.10196352038273697
```

Combining Taggers

One way to address the trade-off between accuracy and coverage is to use the more accurate algorithms when we can, but to fall back on algorithms with wider coverage when necessary. For example, we could combine the results of a bigram tagger, a unigram tagger, and a default tagger, as follows:

- Try tagging the token with the bigram tagger.
- If the bigram tagger is unable to find a tag for the token, try the unigram tagger.
- If the unigram tagger is also unable to find a tag, use a default tagger.

In [23]:
```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(train_sents, backoff=t0)
t2 = nltk.BigramTagger(train_sents, backoff=t1)
t2.evaluate(test_sents)
```

Out[23]:
```
0.8468055417123492
```

Storing trained models

Training a tagger takes time, so we may want to store a trained model into disk for later re-use. Let's save our tagger t2 to a file t2.pkl. The following code will do that.

In [24]:
```
from pickle import dump
output = open('t2.pkl', 'wb')
dump(t2, output, -1)
output.close()
```

We can load the model from the saved file.

In [25]:
```
from pickle import load
input = open('t2.pkl', 'rb')
tagger = load(input)
input.close()
```

Now let's check that it can be used for tagging.

In [26]:
```
text = """The board's action shows what free enterprise
          is up against in our complex maze of regulatory laws ."""
tokens = word_tokenize(text)
print( tagger.tag(tokens) )
```

```
[('The', 'AT'), ('board', 'NN'), ("'s", 'NN'), ('action', 'NN'), ('shows', 'NN
S'), ('what', 'WDT'), ('free', 'JJ'), ('enterprise', 'NN'), ('is', 'BEZ'), ('u
p', 'RP'), ('against', 'IN'), ('in', 'IN'), ('our', 'PP$'), ('complex', 'JJ'),
 ('maze', 'NN'), ('of', 'IN'), ('regulatory', 'NN'), ('laws', 'NNS'), ('.', '.
')]
```

Transformation-based POS Tagging (Brill Tagging)

As we learned in the course "Text Mining", Brill tagging method learn transformation rules from the training data.

In order to train a Brill POS tagger, first we need to create an initial tagger. Here we try the unigram tagger.

In [27]:
```
initial_tagger = nltk.UnigramTagger(train_sents)
initial_tagger.evaluate(test_sents)
```

Out[27]: 0.8129173726701884

The initial tagger, then, is passed to a Brill tagger.

In [28]:
```
from nltk.tbl.template import Template
from nltk.tag.brill import Pos, Word
from nltk.tag import BrillTaggerTrainer

Template._cleartemplates() #clear any templates created in earlier tests
templates = [Template(Pos([-1])), Template(Pos([-1]), Word([0]))]
tt = BrillTaggerTrainer(initial_tagger, templates, trace=3)
brill_tagger = tt.train(train_sents, max_rules=20)
```

```
TBL train (fast) (seqs: 4160; tokens: 90521; tpls: 2; min score: 2; min acc: N
one)
Finding initial useful rules...
    Found 4397 useful rules.

           B      |
   S   F   r   O  |        Score = Fixed - Broken
   c   i   o   t  |  R     Fixed = num tags changed incorrect -> correct
   o   x   k   h  |  u     Broken = num tags changed correct -> incorrect
   r   e   e   e  |  l     Other = num tags changed incorrect -> incorrect
   e   d   n   r  |  e
------------------+-------------------------------------------------------
 104 166  62   3  | NN->VB if Pos:TO@[-1]
  84 111  27   1  | IN->IN-TL if Pos:NN-TL@[-1] & Word:of@[0]
  64  67   3   0  | VBN->VBD if Pos:NP@[-1]
  60  60   0   1  | NN->VB if Pos:MD@[-1]
  53  53   0   0  | VBD->VBN if Pos:BEDZ@[-1]
  45  45   0   2  | VB->NN if Pos:AT@[-1]
  45  48   3   0  | VBN->VBD if Pos:PPS@[-1]
  42  45   3   0  | PPS->PPO if Pos:VB@[-1] & Word:it@[0]
  36  36   0   0  | VBD->VBN if Pos:HVZ@[-1]
  35  37   2   0  | CS->DT if Pos:IN@[-1] & Word:that@[0]
  34  34   0   0  | VBD->VBN if Pos:BE@[-1]
  30  30   0   0  | VB->NN if Pos:JJ@[-1]
  28  36   8   1  | TO->IN if Pos:RP@[-1]
  27  27   0   0  | VBD->VBN if Pos:HVD@[-1]
  24  24   0   0  | VBN->VBD if Pos:WPS@[-1]
  19  27   8   0  | TO->IN if Pos:CD@[-1]
  19  19   0   0  | VBD->VBN if Pos:HV@[-1]
  18  29  11   0  | TO->IN if Pos:IN@[-1]
  18  18   0   0  | VBN->VBD if Pos:PPSS@[-1]
  17  24   7   0  | PPS->PPO if Pos:IN@[-1] & Word:it@[0]
```

Now, we evaluate the trained tagger on the test set.

In [29]:
```
brill_tagger.evaluate(test_sents)
```

Out[29]:
```
0.8221867836140736
```

Classifier based tagging

The `ClassifierBasedPOSTagger` uses classification to do part-of-speech tagging. Features are extracted from words, then passed to an internal classifier. The classifier classifies the features and returns a label; in this case, a part-of-speech tag.

In [30]:
```
from nltk.tag.sequential import ClassifierBasedPOSTagger
tagger = ClassifierBasedPOSTagger(train=train_sents)
tagger.evaluate(test_sents)
```

Out[30]:
```
0.8875710156483604
```

We can customize the classifier.

In [31]:
```
from nltk.classify import MaxentClassifier
# me_tagger = ClassifierBasedPOSTagger(train=train_sents, classifier_builder=MaxentClassifier.train)
# me_tagger.evaluate(test_sents)
```

Custom feature detector

In the classifier based tagging, we can customize the feature extraction. Here we use unigram features by defining a function `unigram_feature_detector`.

In [32]:
```
from nltk.tag.sequential import ClassifierBasedTagger

def unigram_feature_detector(tokens, index, history):
    return {'word': tokens[index]}

tagger = ClassifierBasedTagger(train=train_sents, feature_detector=unigram_feature_detector)
tagger.evaluate(test_sents)
```

Out[32]: 0.8094288846805542

Extracting information from text

The content in the section is a short summary of the chapter 7, "Extracting Information from Text" (http://www.nltk.org/book/ch07.html) of the book *Natural Language Processing with Python*.

Noun Phrase Chunking

In the task, we search for chunks corresponding to individual noun phrases. Base noun phrases do not contain other noun phrases. For example, here is some Wall Street Journal text with NP-chunks marked using brackets:

```
[ The/DT market/NN ] for/IN \[ system-management/NN software/NN \] for/IN \[ Digital/NNP \] \[ 's/POS hardware/NN \] is/VBZ fragmented/JJ enough/RB that/IN [ a/DT giant/NN ] such/JJ as/IN [ Computer/NNP Associates/NNPS ] should/MD do/VB well/RB there/RB ./.
```

There are many ways to extract base noun phrases from text. In this section, we will train a tagger from text data, which can extract noun phrases from text. We will learn about tagging problems in the next lecture of the class.

We will use CoNLL 2000 Corpus. The corpus contain sentences in which noun phrases are annotated in IOB notation. As we have seen, each sentence is represented using multiple lines, as shown below:

```
he PRP B-NP
accepted VBD B-VP
the DT B-NP
position NN I-NP
...
```

In the first step, we load CoNLL 2000 Corpus and print an example sentence.

In [33]:
```
from nltk.corpus import conll2000

print(conll2000.chunked_sents('train.txt')[99])
```

```
(S
  (PP Over/IN)
  (NP a/DT cup/NN)
  (PP of/IN)
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  (VP told/VBD)
  (NP his/PRP$ story/NN)
  ./.)
```

As you can see, the CoNLL 2000 corpus contains three chunk types: NP chunks, which we have already seen; VP chunks such as has already delivered; and PP chunks such as because of. Since we are only interested in the NP chunks right now, we can use the chunk_types argument to select them:

In [34]:
```
print(conll2000.chunked_sents('train.txt', chunk_types=['NP'])[99])
```

```
(S
  Over/IN
  (NP a/DT cup/NN)
  of/IN
  (NP coffee/NN)
  ,/,
  (NP Mr./NNP Stone/NNP)
  told/VBD
  (NP his/PRP$ story/NN)
  ./.)
```

We get the training, test data by using the following code.

In [35]:
```
test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])
train_sents = conll2000.chunked_sents('train.txt', chunk_types=['NP'])
```

Now we define a ChunkParser by using the following code.

In [36]:
```
class ChunkParser(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.TrigramTagger(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.conlltags2tree(conlltags)

NPChunker = ChunkParser(train_sents)
print(NPChunker.evaluate(test_sents))
```

```
ChunkParse score:
    IOB Accuracy:  93.3%
    Precision:     82.5%
    Recall:        86.8%
    F-Measure:     84.6%
```

Now we use the NPChunker to parse a new sentence. First, we need to perform POS tagging on the sentence.

In [37]:
```
sent = 'We saw the yellow dog.'
tagged_sent = nltk.pos_tag(nltk.word_tokenize(sent))
print(tagged_sent)
```

```
[('We', 'PRP'), ('saw', 'VBD'), ('the', 'DT'), ('yellow', 'JJ'), ('dog', 'NN')
, ('.', '.')]
```

Then we apply NPChunker for the POS tagged sentence.

In [38]:
```
print( NPChunker.parse(tagged_sent) )
```

```
(S (NP We/PRP) saw/VBD (NP the/DT yellow/JJ dog/NN) ./.)
```

We get two base noun phrases "We" and "the yellow dog" from the sentence.

Named Entity Recognition

Named entities are definite noun phrases that refer to specific types of individuals, such as organizations, persons, dates, and so on. NLTK provides functions to automatically extract named entities from sentences.

In [39]:
```
from nltk.chunk import ne_chunk
sent = 'My name is Donald Trump'
print( ne_chunk(nltk.pos_tag(nltk.word_tokenize(sent))) )
```

```
(S My/PRP$ name/NN is/VBZ (PERSON Donald/NNP Trump/NNP))
```

Relation extraction

Relation extraction is to extract relations between entities in the sentence. We often extract relations in triple forms (X, α, Y), where X and Y are named entities of the required types, and α is the string of words that intervenes between X and Y. We can use pattern-based methods or machine-learning-based methods.

See the section 6 of the chapter 7 (http://www.nltk.org/book/ch07.html) of the book *Natural Language Processing with Python*.

Analyzing structures of sentences

In this section, we will learn how to perform analyzing structures of sentences. We learn to kinds of sentences' structural analysis:

- Phrasal structural analysis
- Dependency analysis

There are so many good tutorials about using nltk. You may want to refer to chapter 8, Analyzing Sentence Structure (http://www.nltk.org/book/ch08.html) of the book *Natural Language Processing with Python*.

Actually, I did not find begin-to-end tools for analyzing structures of sentences. So, for analyzing structures of sentences, I will suggest Stanford CoreNLP tool (http://stanfordnlp.github.io/CoreNLP/).

References

- Bird, Steven; Klein, Ewan; Loper, Edward (2009). *Natural Language Processing with Python*. http://www.nltk.org/book/ (http://www.nltk.org/book/)
- NLTK in 20 minutes (http://www.slideshare.net/japerk/nltk-in-20-minutes), by Jacob Perkins