SIMS 290-2: Applied Natural Language Processing

Assignment 1

Lecturer: Prof. Marti Hearst *Solution: Preslav Nakov and Barbara Rosario*

1 Tokenizing Exercise 1. Accessing and tokenizing a text file.

1.1 Problem and Solution

Obtain some plain text data (e.g. visit a web-page and save it as plain text), and store it in a file 'corpus.txt'.

(a) Using the `open()` and `read()` functions, load the text into a string variable and print it.

The code below is straightforward. A small fragment of a Web page has been chosen in order to better fit the text (see below).

```
text_str = open('corpus.txt').read()
print '\n(1a) Loading and printing text:'
print text_str
```

(b) Now, initialize a new token with `Token()`, using this text. Tokenize the text with `WhitespaceTokenizer`, and specify that the result should be stored in the `WORDS` property. Print the result.

Again straightforward (in fact copied from the tutorial).

```
from nltk.tokenizer import *
text_token = Token(TEXT=text_str)
WhitespaceTokenizer(SUBTOKENS='WORDS').tokenize(text_token)
print '\n(1b) Tokenized text using the WhitespaceTokenizer:'
print text_token
```

(c) Next, compute the number of tokens, using the `len()` function, and print the result.

```
print '\n(1c) Computing the number of tokens:', len(text_token['WORDS'])
```

(d) Finally, discuss shortcomings of this method for tokenizing text. In particular, identify any material which has not been correctly tokenized. (You may need to look for a more complex text.)

Looking at the program output below, we can see that tokenizing on white space is not a good idea. A list of some observed errors follows:

- The commas are included as part of the token, e.g. `<Sifry,>`
- The fullstops are included as part of the token, e.g. `<July.>`

- The quotes are not handled correctly, e.g. `<"real-time>`, `<"dead>`
- Combinations of the above: `<blagosphere,">`, `<air.">`

But we can get into more trouble if we try to tokenize biomedical text. We probably want the gene/protein name `N-dodecyl-N, N (dimethylammonio) butyrate` to be a single token but this is not the case. It gets even more tricky for some special text section such as citations etc. (See the end of the output below). Note that we can break the long lines into multiple ones using the "\" sign.

```
bio_sentence = 'The receptor-ligand complexes were then released from the cell membrane preparation ' +\
               'by incubation with RIB + detergent, which contained RIB appended with 50 mM ' +\
               'N-dodecyl-N, N (dimethylammonio) butyrate, 1.5% glycerol, and 2% NP-40 for 1 h at 37C' +\
               'and centrifuged at 95,000 rpm for 3 h at 15C.'
text_token = Token(TEXT=bio_sentence)
WhitespaceTokenizer(SUBTOKENS='WORDS').tokenize(text_token)
print '\n(1d) Tokenized biomedical text using WhitespaceTokenizer:'
print bio_sentence
print '\n', text_token

bio_sentence = 'Monboisse J. C., Garnotel R., Bellon G., Ohno N., Perreau C., Borel J. P., Kefalides N. A. ' +\
               'The 3 chain of type IV collagen prevents activation of human polymorphonuclear leukocytes. ' +\
               'J. Biol. Chem., 269: 25475-25482, 1994.[Abstract/Free Full Text] '
text_token = Token(TEXT=bio_sentence)
WhitespaceTokenizer(SUBTOKENS='WORDS').tokenize(text_token)
print '\n(1d) Tokenized biomedical citation text using the WhitespaceTokenizer:'
print bio_sentence
print '\n', text_token
```

1.2 Program Output

```
>>>
(1a) Loading and printing text:
Dave Sifry, founder and CEO of
Technorati - a real-time search engine that tracks the latest
happenings in the blogosphere - gleaned some valuable insights
from working with CNN during the Democratic Convention in July.

Television, he realized, is a moment-to-moment medium that lives
and dies on instant commentary. In comparison, blogging requires a
slightly longer time frame, anywhere between a half hour to a week
to synthesize information and spew it back with a modicum of
analysis. Online news sites and wire service reports, which
usually go through at least one layer of editing, take a little
longer. Newspapers need a day or more, while magazines don't hit
newsstands for weeks, sometimes months.

When CNN producers, who had tapped Technorati to provide
"real-time analysis of the political blogosphere," put Sifry on
the spot and told him to immediately analyze what bloggers were
saying about a particular speech, he found himself facing the
digital equivalent of "dead air." So he was forced to improvise.

(1b) Tokenized text using the WhitespaceTokenizer:
```

```
<[<Dave>,<Sifry,>, <founder>, <and>, <CEO>, <of>, <Technorati>,
<->, <a>, <real-time>, <search>, <engine>, <that>, <tracks>,
<the>, <latest>, <happenings>, <in>, <the>, <blogosphere>, <->,
<gleaned>, <some>, <valuable>, <insights>, <from>, <working>,
<with>, <CNN>, <during>, <the>, <Democratic>, <Convention>, <in>,
<July.>, <Television,>, <he>, <realized,>, <is>, <a>,
<moment-to-moment>, <medium>, <that>, <lives>, <and>, <dies>,
<on>, <instant>, <commentary.>, <In>, <comparison,>, <blogging>,
<requires>, <a>, <slightly>, <longer>, <time>, <frame,>,
<anywhere>, <between>, <a>, <half>, <hour>, <to>, <a>, <week>,
<to>, <synthesize>, <information>, <and>, <spew>, <it>, <back>,
<with>, <a>, <modicum>, <of>, <analysis.>, <Online>, <news>,
<sites>, <and>, <wire>, <service>, <reports,>, <which>, <usually>,
<go>, <through>, <at>, <least>, <one>, <layer>, <of>, <editing,>,
<take>, <a>, <little>, <longer.>, <Newspapers>, <need>, <a>,
<day>, <or>, <more,>, <while>, <magazines>, <don't>, <hit>,
<newsstands>, <for>, <weeks,>, <sometimes>, <months.>, <When>,
<CNN>, <producers,>, <who>, <had>, <tapped>, <Technorati>, <to>,
<provide>, <"real-time>, <analysis>, <of>, <the>, <political>,
<blogosphere,">, <put>, <Sifry>, <on>, <the>, <spot>, <and>,
<told>, <him>, <to>, <immediately>, <analyze>, <what>, <bloggers>,
<were>, <saying>, <about>, <a>, <particular>, <speech,>, <he>,
<found>, <himself>, <facing>, <the>, <digital>, <equivalent>,
<of>, <"dead>, <air.">, <So>, <he>, <was>, <forced>, <to>,
<improvise.>]>

(1c) Computing the number of tokens: 164

(1d) Tokenized biomedical text using WhitespaceTokenizer: The
receptor-ligand complexes were then released from the cell
membrane preparation by incubation with RIB + detergent, which
contained RIB appended with 50 mM N-dodecyl-N, N (dimethylammonio)
butyrate, 1.5% glycerol, and 2% NP-40 for 1 h at 37Cand centrifuged
at 95,000 rpm for 3 h at 15C.

<[<The>, <receptor-ligand>, <complexes>, <were>, <then>, <released>, <from>,
<the>, <cell>, <membrane>, <preparation>, <by>, <incubation>, <with>, <RIB>,
<+>, <detergent,>, <which>, <contained>, <RIB>, <appended>, <with>, <50>, <mM>,
<N-dodecyl-N,>, <N>, <(dimethylammonio)>, <butyrate,>, <1.5%>, <glycerol,>, <and>, <2%>,
<NP-40>, <for>, <1>, <h>, <at>, <37\xb0Cand>, <centrifuged>, <at>,
<95,000>, <rpm>, <for>, <3>, <h>, <at>, <15\xb0C.>]>

(1d) Tokenized biomedical citation text using the
WhitespaceTokenizer: Monboisse J. C., Garnotel R., Bellon G., Ohno
N., Perreau C., Borel J. P., Kefalides N. A. The 3 chain of type
IV collagen prevents activation of human polymorphonuclear
leukocytes. J. Biol. Chem., 269: 25475-25482, 1994.[Abstract/Free
Full Text]

<[<Monboisse>, <J.>, <C.,>, <Garnotel>, <R.,>, <Bellon>, <G.,>,
<Ohno>, <N.,>, <Perreau>, <C.,>, <Borel>, <J.>, <P.,>,
<Kefalides>, <N.>, <A.>, <The>, <3>, <chain>, <of>, <type>, <IV>,
<collagen>, <prevents>, <activation>, <of>, <human>,
<polymorphonuclear>, <leukocytes.>, <J.>, <Biol.>, <Chem.,>,
<269:>, <25475-25482,>, <1994.[Abstract/Free>, <Full>, <Text]>]>
```

```
>>>
```

2 Tokenizing Exercise 2. Tokenizing text using regular expressions.

2.1 Problem and Solution

Obtain some plain text data (e.g. visit a web-page and save it as plain text), and store it in a file 'corpus.txt'. so that you can answer the following questions.

(a) Word processors typically hyphenate words when they are split across a linebreak. When word-processed documents are converted to plain text, the pieces are usually not recombined. It is easy to discover such texts by searching on the web for broken words, e.g. `depart- ment`. Create a `RegexpTokenizer` which treats such broken words as a single token.

Again, copied from the tutorial with minor changes. All we need to do is to add the pattern `r'(\w+(\-\s*\w+)?)'`. You can see in the program output below that it handles correctly broken words such as `depart- ment` and `afore- mentioned`.

```
print '\n(2a) Tokenizer that copes with hyphenated words:'
text_str = open('corpus2.txt').read()
print text_str

from nltk.tokenizer import *
text_token = Token(TEXT=text_str)
regexp = r'(\w+(\-\s*\w+)?)|(\$\d+\.\d+)|([^\w\s]+)'
tokenizer = RegexpTokenizer(regexp, SUBTOKENS='WORDS')
tokenizer.tokenize(text_token)
print text_token
```

If we want to consider the possibility of having a space between the word and the '-', we would have:

```
regexp = r'(\w+(\s*\-\s*\w+)?)|(\$\d+\.\d+)|([^\w\s]+)'
```

If we also want to to remove the '-' and the spaces around it, we need more code. Here we use a regular expression against each single token and perform a substitution, if needed:

```
print '\nNow, if we want to substitute:'
for tok in text_token['WORDS']:
    tok['TEXT']= re.sub('\s*-\s+', '', tok['TEXT'])
print text_token
```

(b) Consider the following book title: *This Is the Beat Generation: New York-San Francisco-Paris*. What would it take to be able to tokenize such strings so that each city name was stored as a single token?

The only way to cope with that problem is to have some world knowledge, i.e. that *New York* and *San Francisco* should go together. Thus we can mark them in some special way during a preprocessing step or as a lexicalized tagger rule looking for these specific city names (and named entities /NE/, in general). There are further problems though. We need no longer treat the dash symbol '-' as an internal token symbol (not

necessarily, because we can add e.g. a special rule that: if we have a NE on either side of the dash, than the dash cannot be an internal symbol). But whatever we decide there will almost certainly be a lot of exceptions and additional issues, especially across domains.

Note that we cannot just say e.g. that we are happy to take two words consecutive words one after the other, if separated by space. Consider the code:

```
print '\n(2b:1) Does not really work. Rather an illustration of the problems:'
text_token = Token(TEXT='This Is the Beat Generation: New York-San Francisco-Paris')
print text_token
regexp = r'(\w+\s\w+)|([^\w\s]+)|(\w+)'
tokenizer = RegexpTokenizer(regexp, SUBTOKENS='WORDS')
tokenizer.tokenize(text_token)
print text_token
```

As the output below shows, it tokenizes correctly the city names but wrongly groups together also "*This Is*" and "*the Beat*".

Here is what happens when we try to fix it quickly using a rule that overfits to the example. It almost works: while it includes the hyphen as part of the token, it can be removed by the code from (2a).

```
print '\n(2b:2) Almost works, but overfits to the current example only:'
regexp = r'(\w+(\s\w+\-)?)'
tokenizer = RegexpTokenizer(regexp, SUBTOKENS='WORDS')
tokenizer.tokenize(text_token)
print text_token
```

Here is a more sophisticated approach which uses lexicalized rules that group together as a city name any two word sequence where the first word is "*New*" or "*San*" (both capitalized).

```
print '\n(2b:3) Uses lexicalized rules:'
import string
wordr = r'(\w+)'
punct = r'([^\w\s]+)'
city = r'((New|San)+\s\w+)'
regexp = string.join([punct,city,wordr],"|")
tokenizer = RegexpTokenizer(regexp, SUBTOKENS='WORDS')
tokenizer.tokenize(text_token)
print text_token
```

2.2 Program Output

```
>>>
(2a) Tokenizer that copes with hyphenated words:
The claimant was injured on September 28, 1946, while in the
employ of the Aberdeen Plywood Corporation. The injury occurred
when the claimant fell a distance of 12 feet while scaling logs,
and according to the report of accident, which was signed by Dr.
Skarperud and filed with the depart- ment of labor and industries
on October 4, 1946, resulted in a fracturing of the terminal end
of the 7th cervical vertebra. By an order dated February 17, 1947,
the claim was closed with no award for permanent partial
```

```
disability and on or about May 6, 1952 the claimant filed with the
department of labor and industries an application to reopen his
claim on the ground of aggravation of condition. This application
for reopening was supported by his then attending physician, Dr.
Dwyer, but on or about May 14, 1952, the department entered an
order denying the claimant's application to reopen on the ground
that the five year statute had run. Shortly after the entry of the
afore- mentioned order Dr. Skarperud wrote the depart- ment
stating that the claimant's condition had become aggravated and
that he was in need of further relief.

<[<The>, <claimant>, <was>, <injured>, <on>, <September>, <28>,
<,>, <1946>, <,>, <while>, <in>, <the>, <employ>, <of>, <the>,
<Aberdeen>, <Plywood>, <Corporation>, <.>, <The>, <injury>,
<occurred>, <when>, <the>, <claimant>, <fell>, <a>, <distance>,
<of>, <12>, <feet>, <while>, <scaling>, <logs>, <,>, <and>,
<according>, <to>, <the>, <report>, <of>, <accident>, <,>,
<which>, <was>, <signed>, <by>, <Dr>, <.>, <Skarperud>, <and>,
<filed>, <with>, <the>, <depart- ment>, <of>, <labor>, <and>,
<industries>, <on>, <October>, <4>, <,>, <1946>, <,>, <resulted>,
<in>, <a>, <fracturing>, <of>, <the>, <terminal>, <end>, <of>,
<the>, <7th>, <cervical>, <vertebra>, <.>, <By>, <an>, <order>,
<dated>, <February>, <17>, <,>, <1947>, <,>, <the>, <claim>,
<was>, <closed>, <with>, <no>, <award>, <for>, <permanent>,
<partial>, <disability>, <and>, <on>, <or>, <about>, <May>, <6>,
<,>, <1952>, <the>, <claimant>, <filed>, <with>, <the>,
<department>, <of>, <labor>, <and>, <industries>, <an>,
<application>, <to>, <reopen>, <his>, <claim>, <on>, <the>,
<ground>, <of>, <aggravation>, <of>, <condition>, <.>, <This>,
<application>, <for>, <reopening>, <was>, <supported>, <by>,
<his>, <then>, <attending>, <physician>, <,>, <Dr>, <.>, <Dwyer>,
<,>, <but>, <on>, <or>, <about>, <May>, <14>, <,>, <1952>, <,>,
<the>, <department>, <entered>, <an>, <order>, <denying>, <the>,
<claimant>, <'>, <s>, <application>, <to>, <reopen>, <on>, <the>,
<ground>, <that>, <the>, <five>, <year>, <statute>, <had>, <run>,
<.>, <Shortly>, <after>, <the>, <entry>, <of>, <the>, <afore-
mentioned>, <order>, <Dr>, <.>, <Skarperud>, <wrote>, <the>,
<depart- ment>, <stating>, <that>, <the>, <claimant>, <'>, <s>,
<condition>, <had>, <become>, <aggravated>, <and>, <that>, <he>,
<was>, <in>, <need>, <of>, <further>, <relief>, <.>]>

Now, if we want to substitute:
<[<The>, <claimant>, <was>, <injured>, <on>, <September>, <28>,
<,>, <1946>, <,>, <while>, <in>, <the>, <employ>, <of>, <the>,
<Aberdeen>, <Plywood>, <Corporation>, <.>, <The>, <injury>,
<occurred>, <when>, <the>, <claimant>, <fell>, <a>, <distance>,
<of>, <12>, <feet>, <while>, <scaling>, <logs>, <,>, <and>,
<according>, <to>, <the>, <report>, <of>, <accident>, <,>,
<which>, <was>, <signed>, <by>, <Dr>, <.>, <Skarperud>, <and>,
<filed>, <with>, <the>, <department>, <of>, <labor>, <and>,
<industries>, <on>, <October>, <4>, <,>, <1946>, <,>, <resulted>,
<in>, <a>, <fracturing>, <of>, <the>, <terminal>, <end>, <of>,
<the>, <7th>, <cervical>, <vertebra>, <.>, <By>, <an>, <order>,
<dated>, <February>, <17>, <,>, <1947>, <,>, <the>, <claim>,
<was>, <closed>, <with>, <no>, <award>, <for>, <permanent>,
```

```
<partial>, <disability>, <and>, <on>, <or>, <about>, <May>, <6>,
<,>, <1952>, <the>, <claimant>, <filed>, <with>, <the>,
<department>, <of>, <labor>, <and>, <industries>, <an>,
<application>, <to>, <reopen>, <his>, <claim>, <on>, <the>,
<ground>, <of>, <aggravation>, <of>, <condition>, <.>, <This>,
<application>, <for>, <reopening>, <was>, <supported>, <by>,
<his>, <then>, <attending>, <physician>, <,>, <Dr>, <.>, <Dwyer>,
<,>, <but>, <on>, <or>, <about>, <May>, <14>, <,>, <1952>, <,>,
<the>, <department>, <entered>, <an>, <order>, <denying>, <the>,
<claimant>, <'>, <s>, <application>, <to>, <reopen>, <on>, <the>,
<ground>, <that>, <the>, <five>, <year>, <statute>, <had>, <run>,
<.>, <Shortly>, <after>, <the>, <entry>, <of>, <the>,
<aforementioned>, <order>, <Dr>, <.>, <Skarperud>, <wrote>, <the>,
<department>, <stating>, <that>, <the>, <claimant>, <'>, <s>,
<condition>, <had>, <become>, <aggravated>, <and>, <that>, <he>,
<was>, <in>, <need>, <of>, <further>, <relief>, <.>]>

(2b:1) Does not really work. Rather an illustration of the problems:
<This Is the Beat Generation: New York-San Francisco-Paris>
<[<This Is>, <the Beat>, <Generation>, <:>, <New York>, <->, <San Francisco>, <->, <Paris>]>

(2b:2) Works, but overfits to the current example only:
<[<This>, <Is>, <the>, <Beat>, <Generation>, <New York->, <San Francisco->, <Paris>]>

(2b:3) Uses lexicalized rules:
<[<This>, <Is>, <the>, <Beat>, <Generation>, <:>, <New York>, <->, <San Francisco>, <->, <Paris>]>
>>>
```

3 Tokenizing Exercise 3. Working with tagged text.

3.1 Problem and Solution

Write a program which loads the Brown corpus, then, given a word, lists the possible tags for the word each with a frequency count. For example, for the word strike the program would generate: `[('nn', 25), ('vb', 21)]`. (Hint: this task involves sorting and reversing a list of tuples which has the form `[(21,'vb'), (25, 'nn')]`. To convert such lists into the required form, use `word_freq = [(y,x) for (x,y) in freq_word]`.)

We use the conditional frequency distribution structure to collect the statistics over the whole Brown corpus. (Note that the terminology is a bit confusing between Count and Frequency in NLTK. They use Count to mean frequency of occurrence of iterms, and Freq to mean what we're calling probability here.) We then convert all tokens to lower case before we start counting them. Finally, we produce the required list using a special function: `getPOSdistr()`.

```
from nltk.corpus import brown
from nltk.probability import ConditionalFreqDist
cfdist = ConditionalFreqDist()
for item in brown.items():
    text_token = brown.read(item)
    for token in text_token['WORDS']:
        tag = token['TAG']
        lowerToken = token['TEXT'].lower()  # normalize the word (i.e. convert to LOWERCASE!)
        cfdist[lowerToken].inc(tag)
```

```
def getPOSdistr(word):
    distr = []
    for POStag in cfdist[word].sorted_samples():
        distr.append((POStag, cfdist[word].count(POStag)))
    return distr

print '\n(3) Getting the probability distribution of the word \'strike\':'
print cfdist['strike']
print getPOSdistr('strike')
```

It is possible to simplify the definitions above, both for the counts and for the probability distribution:

```
def getPOSdistrProb(word):
    return [(i, cfdist[word].freq(i)) for i in cfdist[word].sorted_samples()]

def getPOSdistrCount(word):
    return [(i, cfdist[word].count(i)) for i in cfdist[word].sorted_samples()]
```

(a) Use your program to print the tags and their frequencies for the following words: can, fox, get, lift, like, but, frank, line, interest. Check that you know the meaning of the high-frequency tags.

We use the conditional frequency distribution we have accumulated and we call the `getPOSdistr()` function to get the output in the right format:

```
words = ['can', 'fox', 'get', 'lift', 'like', 'but', 'frank', 'line', 'interest']
print '\n(3a) Getting the frequency distribution of the words: ', words
for word in words:
    print "%10s --> " % word, getPOSdistr(word)
```

(b) Write a program to find the 20 words which have the greatest variety of different possible tags.

We go through all the conditions (i.e. word types) in the conditional frequency distribution, we call the `B()` method to obtain the number of "bins" (i.e. distinct POS tags) and we accumulate the good words in a list. (Note that you didn't have to have found the `B()` method in the API to do this. You could have instead done `len(cfdist[word].samples())` to have the same effect.) Finally, we sort, then we reverse the list (Python sorting is always ascending) and we output the first 20 words only, as required:

```
wordTagFreqs = []
for word in cfdist.conditions():
    wordTagFreqs.append( (cfdist[word].B(), word) )
wordTagFreqs.sort()
wordTagFreqs.reverse()
print '\n(3b) The 20 words which have the greatest variety of different possible tags: '
print wordTagFreqs[:20]
```

(c) Pick words which can be either a noun or a verb (e.g. deal). Guess which is the most likely tag for each word, then check whether you were right.

The code is very similar to what we had for (a). But now we have a different set of words. Now, suppose we guess everything a noun (nn), as this is the most frequent POS category. Then we would get wrong the following words: `check`, `fly`, `guess` and `saw` (See the output below.).

```
words = ['check', 'code', 'deal', 'fly', 'form', 'guess', 'head',
'interest', 'list', 'measure', 'note', 'saw', 'tag', 'trace',
'use'] print '\n(3c) Words that can be either a noun or a verb: ',
words for word in words:
    print "%10s --> " % word, getPOSdistr(word)
```

3.2 Program Output

```
(3) Getting the frequency distribution for the word 'strike':
cfdist['strike']: <FreqDist: 'nn': 25, 'vb': 22, 'vb-tl': 3>
getPOSdistr('strike'): [('nn', 25), ('vb', 22), ('vb-tl', 3)]
getPOSdistrCount('strike'): [('nn', 25), ('vb', 22), ('vb-tl', 3)]

(3) Getting the probability distribution for the word 'strike':
getPOSdistrFreq('strike'): [('nn', 0.5), ('vb', 0.44), ('vb-tl', 0.059999999999999998)]

(3a) Getting the frequency distribution of the words:
['can', 'fox', 'get', 'lift', 'like', 'but', 'frank', 'line', 'interest']
       can -->  [('md', 1758), ('nn', 7), ('md-hl', 2), ('vb', 2), ('md-nc', 1), ('md-tl', 1), ('nil', 1)]
       fox -->  [('nn', 9), ('np', 2), ('nn-tl', 1)]
       get -->  [('vb', 742), ('vb-tl', 4), ('vb-hl', 2), ('vb-nc', 1)]
      lift -->  [('vb', 18), ('nn', 5)]
      like -->  [('cs', 1012), ('vb', 210), ('jj', 36), ('in', 32), ('in-tl', 1), ('vb-hl', 1)]
       but -->  [('cc', 4218), ('in', 131), ('rb', 26), ('cc-hl', 3), ('cc-nc', 3)]
     frank -->  [('np', 45), ('jj', 18), ('nn', 4), ('np-tl', 1)]
      line -->  [('nn', 281), ('nn-tl', 13), ('vb', 4)]
  interest -->  [('nn', 323), ('nn-hl', 3), ('vb', 3), ('nn-tl', 1)]

(3b) The 20 words which have the greatest variety of different possible tags:
[(19, '1'), (15, 'that'), (14, 'a'), (11, 'to'), (10, 'in'), (10, 'home'), (10, '3'),
(9, 'well'), (9, 'right'), (9, 'out'), (9, 'it'), (9, ':'), (8, 'you'), (8, 'set'),
(8, 'round'), (8, 'open'), (8, 'more'), (8, 'long'), (8, 'little'), (8, 'down')]

(3c) Words that can be either a noun or a verb: ['check', 'code', 'deal', 'fly', 'form',
'guess', 'head', 'interest', 'list', 'measure', 'note', 'saw', 'tag', 'trace', 'use']
     check -->  [('vb', 51), ('nn', 36), ('nn-hl', 1)]
      code -->  [('nn', 22), ('nn-tl', 16), ('vb', 1)]
      deal -->  [('nn', 90), ('vb', 41), ('nn-tl', 8), ('vb-nc', 3), ('vb-hl', 1)]
       fly -->  [('vb', 18), ('nn', 15)]
      form -->  [('nn', 300), ('vb', 51), ('nn-tl', 17), ('nn-hl', 1), ('nn-nc', 1)]
     guess -->  [('vb', 53), ('nn', 3)]
      head -->  [('nn', 403), ('vb', 13), ('jjs', 4), ('nn-tl', 2), ('nn-tl-hl', 1), ('nns', 1)]
  interest -->  [('nn', 323), ('nn-hl', 3), ('vb', 3), ('nn-tl', 1)]
      list -->  [('nn', 125), ('vb', 7), ('nn-tl', 1)]
   measure -->  [('nn', 60), ('vb', 28), ('vb-hl', 2), ('nn-tl', 1)]
      note -->  [('nn', 72), ('vb', 53), ('nn-hl', 1), ('vb-hl', 1)]
       saw -->  [('vbd', 338), ('vb', 9), ('nn', 4), ('nn-tl', 1)]
       tag -->  [('nn', 5)]
     trace -->  [('nn', 13), ('vb', 7), ('jj', 2), ('nn-hl', 1)]
       use -->  [('nn', 352), ('vb', 228), ('nn-hl', 9), ('vb-hl', 2)]
>>>
```

4 Tagging Exercise 1. Explorations with part-of-speech tagged corpora.

4.1 Problem and Solution

Tokenize the Brown Corpus and build one or more suitable data structures so that you can answer the following questions.

We read the Brown corpus as above and we keep just two statistics: (1) conditional tag distribution given word type; and (2) unconditional tag distribution. These are enough to relatively easy answer the problem sub-questions.

```
### 1. Read the Brown corpus
from nltk.corpus import brown
from nltk.probability import FreqDist
from nltk.probability import ConditionalFreqDist

### 2. Structures we need for quick answer to the questions
wordTagDist = ConditionalFreqDist() ### (word,tag) frequencies
tagDist     = FreqDist()            ### tag frequencies

### 3. Read the Brown corpus and populate the structures
for item in brown.items():
    text_token = brown.read(item)
    for token in text_token['WORDS']:
        tag = token['TAG']
        lowerToken = token['TEXT'].lower()  # normalize the word (i.e. convert to LOWERCASE!)
        wordTagDist[lowerToken].inc(tag)
        tagDist.inc(tag)
```

(a) What is the most frequent tag? (This is the tag we would want to assign with a `DefaultTagger`)

We just need to call the `max()` method of the unconditional frequency distribution of the tags.

```
print '\n(1a) The most frequent tag is:', tagDist.max()
print tagDist
```

(b) Which word has the greatest number of distinct tags?

This is a simple iteration through the word types and invocation of the `B()` method for each word type.

```
maxTagsCnt = -1
maxTagsWord = 'N/A'
for word in wordTagDist.conditions():
    if wordTagDist[word].B() > maxTagsCnt:
        maxTagsCnt = wordTagDist[word].B()
        maxTagsWord = word
print '\n(1b) The word with the greatest number of distinct tags is:', maxTagsWord
```

(c) What is the ratio of masculine to feminine pronouns?

The key question here is: how we know that a certain word token represents a masculine/feminine pronoun? The simples solution is to consider as masculine only tokens of the kind `he/pps` and `she/pps`. But we also know what are the tags for pronouns in the Brown corpus. So, we then can investigate the table in *Appendix A* and see which pronouns clearly and unambiguously indicate gender. This leaves as with the following token/tag combinations:

- *masculine:* `his/pp$$, himself/ppl, him/ppo, he/pps, he's/pps+bez, he'd/pps+hvd, he's/pps+hvz, he'll/pps+md, he'd/pps+md`
- *feminine:* `hers/pp$$, herself/ppl, her/ppo, she/pps, she's/pps+bez, she'd/pps+hvd, she's/pps+hvz, she'll/pps+md, she'd/pps+md`

We can access the corresponding frequencies directly in the conditional frequency distribution as follows:

```
mascFemRatioHeSheOnly = 1.0 * wordTagDist['he'].count('pps') / wordTagDist['she'].count('pps')
print '\n(1c) The ratio of masculine to feminine pronouns (he/she only!) is:', mascFemRatioHeSheOnly

masc = wordTagDist['his'].count('pp$$') + wordTagDist['himself'].count('ppl') +\
       wordTagDist['him'].count('ppo') + wordTagDist['he'].count('pps') +\
       wordTagDist['he\'s'].count('pps+bez') + wordTagDist['he\'d'].count('pps+hvd') +\
       wordTagDist['he\'s'].count('pps+hvz') + wordTagDist['he\'ll'].count('pps+md') +\
       wordTagDist['he\'d'].count('pps+md')
fem  = wordTagDist['hers'].count('pp$$') + wordTagDist['herself'].count('ppl') +\
       wordTagDist['her'].count('ppo') + wordTagDist['she'].count('pps') +\
       wordTagDist['she\'s'].count('pps+bez') + wordTagDist['she\'d'].count('pps+hvd') +\
       wordTagDist['she\'s'].count('pps+hvz') + wordTagDist['she\'ll'].count('pps+md') +\
       wordTagDist['she\'d'].count('pps+md')
mascFemRatio = 1.0 * masc / fem
print '(1c) The ratio of masculine to feminine pronouns is:', mascFemRatio
```

(d) How many words are ambiguous, in the sense that they appear with at least two tags?

We just need to iterate through the word types and to call the `B()` method. Note that when we divide two integers, we need to first convert one of them to a float, either by multiplying by a float like 1.0, or by using the `float(int)` command. Below we multiply by 100.0 to convert to a percentage and a float simultaneously. We also use the `%0.2f` to specify that we want exactly two decimal digits after the decimal point. And we use `%%` to output the "`%`" sign.

```
totalWordTypesCnt  = 0
ambigWordTypesCnt  = 0
for word in wordTagDist.conditions():
    totalWordTypesCnt = totalWordTypesCnt + 1
    if wordTagDist[word].B() > 1:
        ambigWordTypesCnt = ambigWordTypesCnt + 1
print '\n(1d) Number of ambiguous word types:', ambigWordTypesCnt
print '(1d) Number of all word types (vocabulary size):', totalWordTypesCnt
print '(1d) Percentage of ambiguous vocabulary types: %0.2f%%'\
       % (100.0 * ambigWordTypesCnt / totalWordTypesCnt)
```

(e) What percentage of word occurrences in the Brown Corpus involve these ambiguous words?

Now we need to iterate through the conditional frequency distribution and to collect the counts but this time calling the `N()` method.

```
totalWordTokensCnt  = 0
ambigWordTokensCnt  = 0
for word in wordTagDist.conditions():
    totalWordTokensCnt = totalWordTokensCnt + wordTagDist[word].N()
    if wordTagDist[word].B() > 1:
        ambigWordTokensCnt = ambigWordTokensCnt + wordTagDist[word].N()
print '\n(1e) Percentage of word occurrences in the Brown Corpus involving ambiguous words: %0.2f%%' %\
      (100.0 * ambigWordTokensCnt / totalWordTokensCnt)
```

(f) Which nouns are more common in their plural form than their singular form? (Only consider regular plurals, formed with the -s suffix.)

Similarly to (c), we need to inspect *Appendix A*. When we do so, we can find that the singular/plural nouns are unambiguously distributed in the following Brown corpus tags:

- singular nouns: `nn, np, nr (nn$, nn+bez, nn+hvz, np$, np+bez, nr$)`
- plural nouns: `nns, nps, nrs (nns$, nps$)`

Note that unlike (c), this time the POS tag is enough, i.e. our rules are no longer lexicalized. We decided not to use the categories inside the parentheses, which contain clitics.

```
wordListNN = []
wordListNP = []
wordListNR = []
for word in wordTagDist.conditions():
    if (wordTagDist[word].count('nn') > 0)\
    and (wordTagDist[word + 's'].count('nns') > 0)\
    and (wordTagDist[word].count('nn') < wordTagDist[word + 's'].count('nns') > 0):
        wordListNN.append((word, wordTagDist[word].count('nn'), wordTagDist[word + 's'].count('nns')))
    if (wordTagDist[word].count('np') > 0)\
    and (wordTagDist[word + 's'].count('nps') > 0)\
    and (wordTagDist[word].count('np') < wordTagDist[word + 's'].count('nps') > 0):
        wordListNP.append((word, wordTagDist[word].count('np'), wordTagDist[word + 's'].count('nps')))
    if (wordTagDist[word].count('nr') > 0)\
    and (wordTagDist[word + 's'].count('nrs') > 0)\
    and (wordTagDist[word].count('nr') < wordTagDist[word + 's'].count('nrs') > 0):
        wordListNR.append((word, wordTagDist[word].count('nr'), wordTagDist[word + 's'].count('nrs')))
print '\n(1f) Nouns which are more common in their plural form than their singular form'
print 'NN & NNS:', wordListNN
print 'NP & NPS:', wordListNP
print 'NR & NRS:', wordListNR
```

(g) Produce an alphabetically sorted list of the distinct words tagged as `md`.

This is a simple loop through the word types and collection of the ones with a non-zero frequency for the tag `md`.

```
mdList = []
for word in wordTagDist.conditions():
    if wordTagDist[word].count('md') > 0:
        mdList.append( (word, wordTagDist[word].count('md')) )
mdList.sort()
```

```
print '\n(1g) Here is an alphabetically sorted list of the distinct words tagged as md:'
print mdList
```

(h) Identify words which can be plural nouns or third person singular verbs (e.g. deals).

Here we need an inspection of *Appendix A* again. We decided to recognize plural nouns or third person singular verbs as follows:

- plural nouns: `nns, nps, nrs`
- third person singular verbs: `vbz`

Note that this excludes e.g. the forms of the verb *to be* etc.

```
wordListH = []
for word in wordTagDist.conditions():
    nnCnt  = wordTagDist[word].count('nns') + wordTagDist[word].count('nps') + wordTagDist[word].count('nrs')
    vbzCnt = wordTagDist[word].count('vbz')
    if (nnCnt > 0) and (vbzCnt > 0):
        wordListH.append(word)
print '\n(1h) Identify words which can be plural nouns or third person singular verbs:'
print wordListH
```

4.2 Program Output

```
>>>
(1a) The most frequent tag is: nn
<FreqDist: 'nn': 152393, 'in': 120537, 'at': 97959, 'jj': 64015,
'.': 60638, ',': 58156, 'nns': 55097, 'cc': 37701, 'rb': 36464,
'np': 34476, 'vb': 33693, 'vbn': 29185, 'vbd': 26167, 'cs': 22143,
'pps': 18253, 'vbg': 17893, 'pp$': 16872, 'to': 14918, 'ppss':
13802, 'cd': 13438, 'nn-tl': 13372, 'md': 12431, 'ppo': 11181,
'bez': 10066, 'bedz': 9806, 'ap': 9522, 'dt': 8957, '``': 8837,
"''": 8789, 'ql': 8735, 'vbz': 7373, 'be': 6360, 'rp': 6009,
'wdt': 5539, 'hvd': 4895, '*': 4603, 'wrb': 4509, 'ber': 4379,
'jj-tl': 4103, 'np-tl': 4016, 'hv': 3928, 'wps': 3924, '--': 3405,
'bed': 3282, 'abn': 3010, 'dti': 2921, 'pn': 2573, 'np$': 2565,
'ben': 2470, 'dts': 2435, 'hvz': 2433, ')': 2273, '(': 2264,
'nns-tl': 2226, 'ex': 2164, 'jjr': 1958, 'od': 1933, 'nr': 1566,
':': 1558, 'nn$': 1480, 'in-tl': 1477, 'nn-hl': 1470, 'do': 1353,
'nps': 1275, 'ppl': 1233, 'rbr': 1182, 'dod': 1047, 'jjt': 1005,
'cd-tl': 896, 'md*': 866, 'at-tl': 746, 'abx': 730, 'beg': 686,
'nns-hl': 609, 'uh': 608, '.-hl': 598, 'vbn-tl': 591, 'np-hl':
517, 'in-hl': 508, 'do*': 485, 'ppss+md': 484, 'doz': 467,
'cd-hl': 444, 'pps+bez': 430 ...>

(1b) The word with the greatest number of distinct tags is: 1

(1c) The ratio of masculine to feminine pronouns (he/she only!) is: 3.33951697585
(1c) The ratio of masculine to feminine pronouns is: 3.08272276058

(1d) Number of ambiguous word types: 9601 (1d) Number of all word types (vocabulary size): 49741
```

```
(1d) Percentage of ambiguous vocabulary types: 19.30%

(1e) Percentage of word occurrences in the Brown Corpus involving ambiguous words: 84.17%

(1f) Nouns which are more common in their plural form than their singular form NN & NNS:
[('appropriation', 4, 8), ('projection', 9, 10), ('absolute', 1,
3), ('libertie', 1, 7), ('ligand', 1, 2), ('fin', 2, 5),
('honeybee', 2, 4), ('outburst', 2, 6), ('critter', 1, 3),
('singer', 8, 13), ('plate', 20, 22), ('plaque', 1, 4), ('duffer',
1, 2), ('fabric', 15, 26), ('passenger', 14, 21), ('professional',
6, 10), ('variable', 11, 25), ('crater', 2, 5), ('minute', 43,
193), ('emotion', 34, 43), ('saving', 3, 17), ('periodical', 4,
5), ('wandering', 1, 3), ('allowance', 16, 23), ('advertisement',
2, 3), ('dimension', 14, 30), ('being', 27, 36), ('bubble', 12,
13), ('norm', 7, 23), ('follower', 3, 17), ('colleague', 9, 23),
('gadget', 4, 7), ('vow', 2, 4), ('contributor', 2, 6), ('sock',
3, 7), ('revision', 8, 9), ('maker', 12, 18), ('robot', 1, 3),
('bronchiole', 4, 9), ('prolusion', 3, 4), ('antagonist', 3, 4),
 ...] NR & NRS: []

(1g) Here is an alphabetically sorted list of the distinct words tagged as md:
[("c'n", 1), ('can', 1758), ('colde', 3), ('could', 1598),
('dare', 5), ('kin', 1), ('maht', 1), ('mai', 1), ('may', 1299),
('maye', 2), ('mayst', 1), ('might', 658), ('must', 999), ('need',
38), ('ought', 68), ('shall', 266), ('should', 883), ('shuld', 3),
('shulde', 1), ('wil', 1), ('will', 2130), ('wilt', 1), ('wod',
1), ('wold', 1), ('wolde', 1), ('would', 2710)]

(1h) Identify words which can be plural nouns or third person singular verbs:
['shocks', 'kids', 'purges', 'marches', 'stays', 'plunges',
'mirrors', 'wants', 'excuses', 'sums', 'gestures', 'watches',
'causes', 'displays', 'replies', 'presses', 'petitions', 'makes',
'interests', 'dishes', 'places', 'searches', 'equals', 'reaches',
'feeds', 'dogs', 'subjects', 'runs', 'encounters', 'turns',
'shares', 'bangs', 'flares', 'spreads', 'challenges', 'bugs',
'hurts', 'conducts', 'houses', 'boils', 'pictures', 'lives',
'fears', 'smiles', 'times', 'lists', 'hits', 'limits', 'likes',
'moderates', 'kisses', 'beats', 'functions', 'crops', 'skins',
'meets', 'spans', 'concentrates', 'preserves', 'mentions',
...]
>>>
```

5 Tagging Exercise 2. Regular Expression Tagging.

5.1 Problem and Solution

We defined the `NN_CD_Tagger`, which can be used as a fall-back tagger for unknown words. This tagger only checks for cardinal numbers. By testing for particular prefix or suffix strings, it should be possible to guess other tags. For example, we could tag any word that ends with `-s` as a plural noun.

First, we need to fetch the Brown corpus. We use the first 10 parts only as it takes too long to load it entirely. We can also easily run out of memory.

```
from nltk.corpus import brown
from nltk.tagger import *
brown_tokens = []
for item in brown.items()[:10]:    # texts 0-9
    brown_tokens.append(brown.read(item))
```

(a) Define a `RegexpTagger` which tests for at least five other patterns in the spelling of words. (Use inline documentation to explain the rules.)

We use mostly suffix rules. We also saw that identifying the commas (a lexicalized rule) has a major impact.

```
myRegexpTagger = RegexpTagger([(r'^[0-9]+(.[0-9]+)?$', 'cd'), (r',', ','), (r'\.', '\.'),\
                               (r'^[A-Z].*\'s$', 'np$'), (r'^[a-z].*\'s$', 'nn$'), (r'^.*ly$', 'rb'),\
                               (r'^.*al$', 'jj'), (r'^.*ful$', 'jj'), (r'^.*ous$', 'jj'),\
                               (r'^.*tion$', 'nn'), (r'^.*ing$', 'vbg'), (r'^.*ble$', 'jj'),\
                               (r'^.*ize$', 'vb'), (r'^.*ed$', 'vbn'), (r'^.*s$', 'nns'),\
                               (r'.*', 'nn')], SUBTOKENS='WORDS')
```

(b) Evaluate the tagger using `tagger_accuracy()`, and discuss your findings.

This is straightforward. The results show we doubled the accuracy of the original tagger, which was about 14%.

```
print '\n(2b) The tagger\'s accuracy is:', tagger_accuracy(myRegexpTagger, brown_tokens)
```

5.2 Program Output

```
>>>
(2b) The tagger's accuracy is: 0.288737717309
>>>
```

6 Tagging Exercise 3. Unigram Tagging.

6.1 Problem and Solution

Train a unigram tagger and run it on some new text.

Copied from the tutorial:

```
from nltk.corpus import brown
from nltk.tagger import *
train_tokens = []
for item in brown.items()[:10]:    # texts 0-9
    train_tokens.append(brown.read(item))

test_tokens = []
for item in brown.items()[10:12]:    # texts 10-11
    test_tokens.append(brown.read(item))
```

(a) Evaluate the tagger as before and discuss your findings.

We just copied from the tutorial with minor modifications. We train the unigram tagger on texts 0-9 and we test on 10-11.

```
uniTagger = UnigramTagger(SUBTOKENS='WORDS')
for tok in train_tokens:
    uniTagger.train(tok)
print '\n(3a) The tagger\'s TRAINING accuracy is: %0.2f%%' % (100.0 * tagger_accuracy(uniTagger, train_tokens))
print '(3a) The tagger\'s TESTING  accuracy is: %0.2f%%' % (100.0 * tagger_accuracy(uniTagger, test_tokens))
```

(b) Observe that some words are not assigned a tag. Why not?

Because they are unknown.

Note! This time the accuracy is much higher, even though the context is not used at all. This is the classic baseline for tagging.

6.2 Program Output

```
>>>
(3a) The tagger's TRAINING accuracy is: 94.80%
(3a) The tagger's TESTING  accuracy is: 64.59%
>>>
```

7 Tagging Exercise 5. Combining taggers with BackoffTagger.

7.1 Problem and Solution

There is typically a trade-off between the accuracy and coverage for taggers: taggers that use more specific contexts usually produce more accurate results, when they have seen those contexts in the training data; but because the training data is limited, they are less likely to encounter each context. The `BackoffTagger` addresses this problem by trying taggers with more specific contexts first; and falling back to the more general taggers when necessary. In this exercise, we examine the effects of using `BackoffTagger`. Create a `DefaultTagger` or a `RegexpTagger`, and a `UnigramTagger`, and a `NthOrderTagger`. Train the `UnigramTagger` and the `NthOrderTagger` using part of the Brown corpus.

As above, we first fetch the Brown corpus. We train the unigram tagger on texts 0-9 and we test on 10-11.

```
from nltk.corpus import brown from nltk.tagger import *
train_tokens = [] for item in brown.items()[:10]:    # texts 0-9
    train_tokens.append(brown.read(item))

test_tokens = [] for item in brown.items()[10:12]:    # texts 10-11
    test_tokens.append(brown.read(item))
```

(a) Evaluate each tagger on unseen data from the Brown corpus. Record the accuracy of the tagger (the percentage of tokens that are correctly tagged). Be sure to use a different section of the corpus for testing than you used for training.

The code below is from the tutorial with minor adaptations. It should be clear enough.

```
# Train the taggers
subtagger1 = RegexpTagger([(r'^[0-9]+(.[0-9]+)?$', 'cd'), (r'.*', 'nn')], SUBTOKENS='WORDS')
subtagger2 = UnigramTagger(SUBTOKENS='WORDS')           # zeroth order tagger
subtagger3 = NthOrderTagger(1, SUBTOKENS='WORDS')       # first order tagger
subtagger4 = NthOrderTagger(2, SUBTOKENS='WORDS')       # second order tagger
subtagger5 = NthOrderTagger(3, SUBTOKENS='WORDS')       # third order tagger
subtagger6 = NthOrderTagger(4, SUBTOKENS='WORDS')       # fourth order tagger

# Train the taggers
for tok in train_tokens:
    subtagger2.train(tok)
    subtagger3.train(tok)
    subtagger4.train(tok)
    subtagger5.train(tok)
    subtagger6.train(tok)

# Output the training accuracy results
print '\n(5a) TRAINING accuracy:'
print 'RegExp tagger  : %5.2f%%' % (100.0 * tagger_accuracy(subtagger1, train_tokens))
print 'Unigram tagger : %5.2f%%' % (100.0 * tagger_accuracy(subtagger2, train_tokens))
print 'Bigram tagger  : %5.2f%%' % (100.0 * tagger_accuracy(subtagger3, train_tokens))
print 'Trigram tagger : %5.2f%%' % (100.0 * tagger_accuracy(subtagger4, train_tokens))
print 'Fourgram tagger: %5.2f%%' % (100.0 * tagger_accuracy(subtagger5, train_tokens))
print 'Fivegram tagger: %5.2f%%' % (100.0 * tagger_accuracy(subtagger6, train_tokens))

print '\n(5a) TESTING accuracy:'
print 'RegExp tagger  : %5.2f%%' % (100.0 * tagger_accuracy(subtagger1, test_tokens))
print 'Unigram tagger : %5.2f%%' % (100.0 * tagger_accuracy(subtagger2, test_tokens))
print 'Bigram tagger  : %5.2f%%' % (100.0 * tagger_accuracy(subtagger3, test_tokens))
print 'Trigram tagger : %5.2f%%' % (100.0 * tagger_accuracy(subtagger4, test_tokens))
print 'Fourgram tagger: %5.2f%%' % (100.0 * tagger_accuracy(subtagger5, test_tokens))
print 'Fivegram tagger: %5.2f%%' % (100.0 * tagger_accuracy(subtagger6, test_tokens))
```

(b) Use `BackoffTagger` to create three different combinations of the basic taggers. Test the accuracy of each combined tagger. Which combinations give the most improvement?

Straightforward. Should be clear from the code and the program output. The more N-gram models we add, the better the result.

```
# Combine the taggers
tagger1 = BackoffTagger([subtagger1], SUBTOKENS='WORDS')
tagger2 = BackoffTagger([subtagger2, subtagger1], SUBTOKENS='WORDS')
tagger3 = BackoffTagger([subtagger3, subtagger2, subtagger1], SUBTOKENS='WORDS')
tagger4 = BackoffTagger([subtagger4, subtagger3, subtagger2, subtagger1], SUBTOKENS='WORDS')
tagger5 = BackoffTagger([subtagger5, subtagger4, subtagger3, subtagger2, subtagger1],\
    SUBTOKENS='WORDS')
tagger6 = BackoffTagger([subtagger6, subtagger5, subtagger4, subtagger3, subtagger2, \
    subtagger1], SUBTOKENS='WORDS')

# Output the training accuracy results
print '\n(5b) TRAINING accuracy:'
print 'RegExp tagger                                          : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger1, train_tokens))
print 'Unigram+RegExp tagger                                  : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger2, train_tokens))
```

```
print 'Bigram+Unigram+RegExp tagger                            : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger3, train_tokens))
print 'Trigram+Bigram+Unigram+RegExp tagger                    : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger4, train_tokens))
print 'Fourgram+Trigram+Bigram+Unigram+RegExp tagger           : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger5, train_tokens))
print 'Fivegram+Fourgram+Trigram+Bigram+Unigram+RegExp tagger : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger6, train_tokens))

# Output the training accuracy results
print '\n(5b) TESTING accuracy:'
print 'RegExp tagger                                          : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger1, test_tokens))
print 'Unigram+RegExp tagger                                  : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger2, test_tokens))
print 'Bigram+Unigram+RegExp tagger                           : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger3, test_tokens))
print 'Trigram+Bigram+Unigram+RegExp tagger                   : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger4, test_tokens))
print 'Fourgram+Trigram+Bigram+Unigram+RegExp tagger          : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger5, test_tokens))
print 'Fivegram+Fourgram+Trigram+Bigram+Unigram+RegExp tagger : %5.2f%%' \
    % (100.0 * tagger_accuracy(tagger6, test_tokens))
```

7.2 Program Output

We see that all taggers significantly outperform the simple `RegExp` tagger. We also see that as we add higher order N-gram taggers the accuracy keeps improving. This improvement is insignificant though, compared to the bigram backoff model. Even the bigram backoff model improves only very slightly over the unigram. We should not take these results too seriously though as we did not use enough data for training and testing. Anyway, as a general rule, as we add more data, the testing accuracy will grow. But it will, of course, always stay below the training accuracy.

```
>>>
(5a) TRAINING accuracy:
RegExp tagger  : 14.91%
Unigram tagger : 94.80%
Bigram tagger  :  0.00%
Trigram tagger :  0.00%
Fourgram tagger:  0.00%
Fivegram tagger:  0.00%

(5a) TESTING accuracy:
RegExp tagger  : 14.94%
Unigram tagger : 64.59%
Bigram tagger  :  0.00%
Trigram tagger :  0.00%
Fourgram tagger:  0.00%
Fivegram tagger:  0.00%

(5b) TRAINING accuracy:
RegExp tagger                                          : 14.91%
Unigram+RegExp tagger                                  : 94.80%
```

```
Bigram+Unigram+RegExp tagger                              : 98.17%
Trigram+Bigram+Unigram+RegExp tagger                      : 99.08%
Fourgram+Trigram+Bigram+Unigram+RegExp tagger             : 99.53%
Fivegram+Fourgram+Trigram+Bigram+Unigram+RegExp tagger : 99.86%

(5b) TESTING accuracy:
RegExp tagger                                             : 14.94%
Unigram+RegExp tagger                                     : 71.89%
Bigram+Unigram+RegExp tagger                              : 72.57%
Trigram+Bigram+Unigram+RegExp tagger                      : 72.68%
Fourgram+Trigram+Bigram+Unigram+RegExp tagger             : 72.70%
Fivegram+Fourgram+Trigram+Bigram+Unigram+RegExp tagger : 72.72%
>>>
```

And here is the result when we use the complex `myRegexpTagger` tagger instead of `tagger1`:

```
>>>
(5b) TRAINING accuracy:
myRegexpTagger tagger                                          : 28.87%
Unigram+myRegexpTagger tagger                                  : 94.80%
Bigram+Unigram+myRegexpTagger tagger                           : 98.17%
Trigram+Bigram+Unigram+myRegexpTagger tagger                   : 99.08%
Fourgram+Trigram+Bigram+Unigram+myRegexpTagger tagger          : 99.53%
Fivegram+Fourgram+Trigram+Bigram+Unigram+myRegexpTagger tagger : 99.86%

(5b) TESTING accuracy:
myRegexpTagger tagger                                          : 26.26%
Unigram+myRegexpTagger tagger                                  : 76.27%
Bigram+Unigram+myRegexpTagger tagger                           : 76.90%
Trigram+Bigram+Unigram+myRegexpTagger tagger                   : 77.01%
Fourgram+Trigram+Bigram+Unigram+myRegexpTagger tagger          : 76.94%
Fivegram+Fourgram+Trigram+Bigram+Unigram+myRegexpTagger tagger : 76.96%
>>>
```

Interestingly, the best accuracy on the *testing* set this time is achieved not for the fivegram tagger but for the trigram one (which is also better than the fourgram one). Of course, this is not the case on the *training* set. Do you have an explanation for this? (BTW, real world taggers are typically based on bigrams, more rearly on trigram, and almost never make use of higher order N-grams.)