Inf2A: Assignment 2

Words, Sentences, Parts of Speech, Grammar

Issued 24 October (Week 6)

Please read through the entire assignment before you start. If you have any questions on the python aspects of the assignment, please ask a demonstrator during one of the lab sessions.

The deadline for this assignment is **12 noon, Monday 7 November**. You should spend no more than 20 hours on it.

Download `2011_inf2a_a01_code.tar` from

`http://www.inf.ed.ac.uk/teaching/courses/inf2a/assignments/`

and untar it (`tar -xvf 2011_inf2a_a01_code.tar`). You should find the data file `nonVBG.py`, the initial grammar file `mygrammar.cfg` which you are asked to extend in Section 4, and the other files in which to put your scripts and comments `tts.py`, `pos.py`, and `parsing.py`.

Submit your work using the following command:

```
submit inf2 inf2a A2 tts.py pos.py parsing.py mygrammar.cfg
```

For writing the requested python scripts using NLTK, you may find helpful the python labs sheets (week 3 and 4) and the NLTK lab sheet (weeks 5 and 6). These are available at `http://www.inf.ed.ac.uk/teaching/courses/inf2a/tutorials/`. Also helpful is the book Natural Language Processing in Python (`http://www.nltk.org/book`).

1 Introduction

For this exercise, you should import the following into your python shell:

- `from nltk.corpus import inaugural, stopwords`
- `from nltk import FreqDist, ConditionalFreqDist, pos_tag, data`
- `import re`
- `from nonVBG import *`

`nonVBG.py` is contained in `2011_inf2a_a01_code.tar`. It initializes the list `nonVBG` used in Section 3.

Summary information about some of the above can be found in the NLTK book:

- Chapter 1, Table 1-2 (Functions defined for Frequency Distributions[1])
- Chapter 2, Table 2-3 (Basic Corpus Functionality[2])
- Chapter 3: Table 3-3 (Basic Regular Expression Meta-Characters[3])

Here are a few further python hints:

1. To get an idea of what has been imported by the above commands and how to access it, try typing

```
>>> inaugural.fileids()
>>> inaugural.words('2009-Obama.txt')
>>> inaugural.sents('2005-Bush.txt')
```

2. In python, all empty things (e.g., `''`, `[]`, `{}`) are of type `None`, which is `False`. Conversely, all non-empty things are `True`. This allows you to replace:

```
if re.match(pattern2,words[i]) != None:

if re.match(...) == None:
```

[1] `http://nltk.googlecode.com/svn/trunk/doc/book/ch01.html#tab-freqdist`
[2] `http://nltk.googlecode.com/svn/trunk/doc/book/ch02.html#tab-corpus`
[3] `http://nltk.googlecode.com/svn/trunk/doc/book/ch03.html#tab-regexp-meta-characters1`

with, respectively,

```
if re.match(pattern2,words[i]):
```

```
if not re.match(...):
```

3. Instead of looping over list indices

```
for i in range(len(words)):
    word = words[i]
```

where `words` is a list, python allows:

```
for word in words:
```

You can loop/iterate over all kinds of objects: strings (char-by-char), lists, and dictionaries. For the latter, the iterator will be the key, as in

```
for key in dict:
    val = dict[key]
```

4. You may find it convenient to use python list comprehensions in the assignment. According to `http://docs.python.org/tutorial/datastructures.html`, a list comprehension consists of an expression followed by a `for` clause, and then zero or more `for` and/or `if` clauses. The result will be a list resulting from evaluating the expression in the context of those `for` and `if` clauses.

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [3*x for x in vec1]
>>> vec2
[6, 12, 18]
>>> [3*x for x in vec1 if x > 3]
[12, 18]
>>> [3*x for x in vec1 if x < 2]
[]
```

5. `for` loops iterate on copies of objects, not references to them. Thus in

```
>>> words = ['dog','house','phantasmagoric']
>>> for word in words:
        if word == 'dog':
            word = 'cat'
```

`words[0]` will not be changed to `'cat'`. Instead, to do this, the following is needed:

```
>>> for word in words:
>>>     if word == 'dog':
            words[words.index(word)] = 'cat'
```

2 Change in words and sentences over time (40 marks)

NLTKs Inaugural Address Corpus (`from nltk.corpus import inaugural`) contains 56 texts, one for each US Presidential Inauguration. These take place once every four years. The inaugural corpus starts with Washington's first inaugural address (from January 1789), and ends with Obama's inaugural address (from January 2009).

This part of the assignment asks you to examine some ways in which language use has changed over this more than 200 year period. To this end, you will look at such properties as the vocabulary commonly used in the addresses, the average length of the sentences making up an address, and what the president is saying about himself.

Your first task is to partition the inaugural addresses into overlapping 20-year time periods, each containing 5 inaugural addresses.

1. Write a python/NLTK script that maps the list of inaugural addresses into a list of 5-element lists. The first sublist should contain the first five elements `['1789-Washington.txt', ..., '1805-Jefferson.txt']`. Each subsequent list should contain the final four elements from the preceding list, plus the next inaugural address — e.g. `['1793-Washington.txt', ... '1809-Madison.txt']`. The final sublist should contain the final five inaugural addresses. Store the complete list of lists in a variable called `inaug20`.

   Add your script to `tts.py`.

2. The next task is to look at characteristics of the inaugural addresses from different time periods and see if and how the words they most frequently use have changed.

   Write a function `word_fdist (inaug_list)` which takes a list of inaugural address names as its only parameter, and returns a *frequency distribution* of the words in those addresses (that is, an object of the class `FreqDist`.) Words that are capitalized should be treated like their lower-case counterparts by using the method `lower` before computing a frequency distribution. You should also exclude short function words like `the`, `and` and `is`, and punctuation marks like '.', ' ?' and ';'. The expression

   ```
   stopwords.words("english")
   ```

   evaluates to a list of short function words in English which you should remove; the list of punctuation marks, you'll have to define yourself. [**Note:** The above expression actually returns a list of *unicode* strings. To avoid certain problems later on, it is strongly recommended that you convert it at an early stage to a list of ascii strings. If `w` is a unicode string, you can obtain the corresponding ascii string as `w.encode()` .]

   Add your function definition to `tts.py`.

   Now define a function `print_most_common()` that computes and prints out, in any reasonable format, the 20 most common words in each (overlapping) 20-year period on your list `inaug20`. Again, add the function definition to `tts.py`.

   (You might find it interesting to reflect on how the most frequent words have changed over time, what might account for this, and what if anything we can conclude from it. However, this does not form part of the assignment.)

3. The next task looks at sentences, to see if their complexity has changed over time. While sentence complexity is properly measured in terms of syntactic features, sentence *length* (i.e., the number of words a sentence contains) is much simpler to compute and not a bad proxy.

   Write a function `sent_length_fdist (inaug_list)` that, given a list of inaugural addresses, returns a frequency distribution of the lengths of the sentences in those addresses (i.e. an object of class `FreqDist`.) Again, remove punctuation from the sentences, but not stopwords this time. Add your script to `tts.py`.

   Now write a function `print_average_lengths()` which computes and prints the *average* sentence length in each of the overlapping 20-year periods. (Try it out to check that it works.) Again, add your script to `tts.py`.

4. Your final task for this section can be seen as an attempt to shed light on what US presidents see themselves as doing or having, what they see as being done to them, and whether this has changed over time. We'll do this by examining phrases that start with "I" or "my" or that end with "me". This can be done using a *conditional frequency distribution*, conditioned on a pair of conditions (the 20-year period and the pronoun of interest).

   Write a function `build_cond_fdist()` that builds and returns a *conditional frequency distribution* (i.e., an object of class `ConditionalFreqDist`) for the words which follow "I" or "my" or precede "me". The conditions for this distribution will be pairs `(period, pronoun)`, where `period` is an *integer* representing the start of the period (e.g. `1789`), and `pronoun` is one of the strings `'I'`, `'my'`, `'me'`.

   The samples in the distribution will be words found before "me" or after "I" or "my", as appropriate. Don't count the short stopwords that featured in Task 2.

   Add this script to `tts.py`.

   Finally, write a function `print_Imyme_words(CFD)` which, given a conditional frequency distribution as above, prints out, keyed by pronoun and then by 20-year period, the words that occur more than once in the appropriate frequency distribution, in order of decreasing frequency. You can check that it works by applying it to the output of `build_cond_fdist`. Again, add your script to `tts.py`.

3 Part-of-speech (POS) tagging (25 marks)

Any POS-tagger used in an industrial-grade system or even a research context will have a *back-off* strategy to deal with words that it doesn't know. While we don't have much information about NLTK's current best POS-tagger, `pos_tag`, which comes pre-trained on the 1-million word *Wall Street Journal Corpus*, it does assign a tag to every word in a text it is given as input.

```
>>> text = 'He has been suffering for three days'.split()
>>> pos_tag(text)
[('He', 'PRP'), ('has', 'VBZ'), ('been', 'VBN'), ('suffering', 'VBG'),
('for', 'IN'), ('three', 'CD'), ('days', 'NNS')]
>>> text = 'He has been dog-fooding for three days'.split()
>>> pos_tag(text)
[('He', 'PRP'), ('has', 'VBZ'), ('been', 'VBN'), ('dog-fooding', 'JJ'),
('for', 'IN'), ('three', 'CD'), ('days', 'NNS')]
```

The above POS-tags are explained in Jurafsky & Martin (2nd ed.), Figure 5.6 (p. 165).

While `pos_tag` works pretty well on certain texts, it does make errors, including errors on some words ending with "ing" like dog-fooding. Some of these words, such as *something* and *string* are nouns (and as such, should be tagged `NN`), and at least one, *during*, is a preposition (part-of-speech tag `IN`). On the other hand, words such as *thinking*, *executing*, and *dog-fooding* are verb forms, which should be part-of-speech tagged as `VBG`. Here your task will be to correct some of these errors automatically, on a second pass through a text that has been part-of-speech tagged by `pos_tag`.

1. Write a function `tag_sentences (sent_list)` which, given a list of sentences (of the kind output by `inaugural.sents('2009-Obama.txt')`), returns a corresponding list of tagged sentences. Add your definition to `pos.py`.

   Now write a function `build_ing_dict (tag_sent_list)` which, given a list of tagged sentences (of the kind output by the previous function), identifies the words ending in "ing". You should tools in the `re` module for this purpose (even though in this instance it would be just as simple to do it without). For each such word, you should record in a dictionary the part-of-speech tags it has been assigned in the text. (See the second python lab sheet for information on dictionaries.) The dictionary should be keyed by words ending in "ing" (as ordinary strings), and the values should be *lists* of part-of-speech tags, to allow for the possibility that the same word might be tagged differently in two places (as sometimes happens). Add your script to `pos.py`.

   You may test your script on Washington's first inaugural address and Obama's inaugural address.

2. In a sentence like "Impress us by climbing Ben Nevis", the verb phrase (`VP`) following "by" indicates *how* the VP preceding the "by" should be (or was) accomplished. As such, its head (here, the word *climbing*) should be tagged as a verb — in particular, `VBG`. For some reason that I have not yet fathomed, `pos_tag` does not always do this.

```
>>> text = 'Impress us by climbing Ben Nevis'.split()
>>> pos_tag(text)
[('Impress', 'NN'), ('us', 'PRP'), ('by', 'IN'),
('climbing', 'VBG'), ('Ben', 'NNP'), ('Nevis', 'NNP')]
>>> text = 'Cultivate peace by observing justice'.split()
```

```
>>> pos_tag(text)
[('Cultivate', 'NNP'), ('peace', 'NN'), ('by', 'IN'),
('observing', 'NN'), ('justice', 'NN')]
```

Write a function `retag_sentence (tag_sent)` that corrects such errors: Given a single tagged sentence (i.e., a sequence of word-tag pairs), output another tagged sentence in which each word ending in "ing" is re-tagged to `VBG` if it follows a preposition (i.e., a word tagged `IN`), and if it is not on the wordlist `nonVBG` (from the .tar file you downloaded). Add this script to `pos.py`.

Test your script on a few inaugural addresses. You should find that at least some improvements to the tagging have been made.

4 Grammar writing and parsing (35 marks)

Here the goal is to write a small grammar that covers a set of short sentences found in the inaugural addresses, and to test it by parsing the sentences according to your grammar, using one of the NLTK parsers. We focus on short sentences because the amount of ambiguity that a non-probabilistic parser can find in a long sentence can be staggering. We will take a short sentence to be one with 8 or fewer tokens, including punctuation. (Because of final punctuation, this means at most 7 genuine words.

1. Write a python/NLTK script to extract the short sentences from the inaugural corpus and pos-tag them. (You need not submit the script you use to do this.)

   Manually identify a subset of at least eight (8) of these sentences whose part-of-speech tagging indicates that they conform (at their top level) to the pattern `S-->NP VP`. (That is, they consist of a *noun phrase*, followed by a *verb phrase* containing a verb and zero or more complements.)

   Look at the context-free grammar given in `mygrammar.cfg`. The commands in `parsing.py` show how such a grammar may be loaded and used to parse a sentence.

   Edit the file to give a context-free grammar which covers these sentences from the root `S` down to the lexical items. Remember that the rules in `mygrammar.cfg` need to specify how a part-of-speech like `NN` rewrites to the relevant lexical items in your sentences, as well as specifying how non-terminal elements like `NP` and `VP` rewrite to a combination of non-terminals and parts-of-speech. You can use rules from grammars in the lecture or lab notes or from the NLTK book, or you can come up with your own. Try to avoid rules that only cover a single phrase.

At least some of your rules should involve *recursion*, either direct or indirect, even though this may not be needed for the sentences in your set.

You may delete the lines marked 'to be deleted'. Make sure that the grammar you want to submit is in `mygrammar.cfg`.

Finally, edit the file `parsing.py` adding a line to parse each of the eight short sentences you have identified, and check that they parse correctly. If none of your sentences involves recursion, add one of your own that does (and mark it as not coming from the corpus).

Again, you should delete the lines marked 'to be deleted'. Make sure your submission is in `mygrammar.cfg`.

5 Plagiarism

There are some general principles in preparing any work:

- You should complete the work yourself, using your own words, code, figures, etc.
- Acknowledge your sources for text, code, figures etc that are not your own.
- Take reasonable precautions to ensure that others do not copy your work and present it as their own.

If you produce your work own unassisted, you are in no danger of plagiarising. If you require some help or collaboration, follow the guidelines below.

It is acceptable to use, without acknowledgement, general ideas picked up in discussion with others or from textbooks. If you write the work you submit in its entirety, you do not need to include an acknowledgement. An exception is when a really pivotal idea comes from an external source, in which case you should acknowledge that source.

There are occasions on which you will wish to include work written by somebody else within your own work. For example, you may want to quote a passage from a book within an essay, or you may be unable to finish a program yourself and want to include a piece of code written by someone else. On such occasions, you should copy the original work verbatim, together with a full and explicit acknowledgement of the original source. The beginning and end of the copied material must be clearly indicated:

- In the case of text copied into an essay or in answer to a question, enclose the text within quotation marks and explicitly indicate its source close to where the copied material appears.

- In the case of code, insert a comment before the start of the code you have copied or collaborated on, indicating who you have got it from or who you have collaborated with in developing it, and then another comment after the end of the relevant code.

These rules about acknowledging the source of work that is not fully your own apply irrespective of the type of work it is (prose, code, etc.). If you collaborate on **anything**, the collaboration must be explicitly acknowledged, and its extent should be stated as fully as possible by all parties involved.

It is **never acceptable** to attempt to disguise someone else's work in order to present it as your own, for example by making cosmetic changes to the text, or by changing variable names or altering comments in code.

When in doubt, state explicitly what you have done and the sources of any help (including fellow students, tutors or textbooks) you receive. This will enable your work to be marked appropriately, and will ensure that you avoid suspicion of plagiarism.

In order to prevent others from copying your work and presenting it as their own, set the protection on your practical work directories appropriately. It is also sensible to ensure that any practical submissions that you print are collected promptly from the printers. By all means discuss your general ideas with other students, but do not distribute or lend your solutions to them.