Lab 2: POS Tagging

Use the html version[1] of this lab if you want to easily access the links or copy and paste commands. Or use the pdf version[2] if you want nicer formatting or a printable sheet.

Goals and motivation of this lab, and what to submit

Tagging is one of the basic steps in developing many Natural Language Processing (NLP) tools, and is often also a first step in starting to annotate and analyze a corpus in a new language. In this lab, we will explore POS tagging and build a (very!) simple POS tagger using an already annotated corpus, just to get you thinking about some of the issues involved.

In addition, this lab demonstrates some basic functions of the NLTK[3] library. NLTK (Natural Language Toolkit) is a popular library for language processing tasks which is developed in Python. It has several useful packages for tasks like tokenization, tagging, parsing, etc. In this lab, we use only very basic functions like loading the data and reading sentences, but we hope you may be inspired to look into additional aspects of NLTK on your own.

As before, we have written most of the code for the lab already and included a lot of explanatory comments, but we will ask you to add a few things here and there. The 'Going Further' section will give you a chance to explore some more advanced topics if you finish early, or later on your own.

At the end of today's lab session, please email me a short message including the following information (If you are working with a partner, only one email is needed for both of you):

- your name and your partner's name;
- a few sentences answering Q4 and Q8 (or if you didn't get to Q8, then send your answer to the last question you did get to).

I will post solutions to each lab but will not provide individual feedback, I just want to make sure that you are actually making an effort.

Preliminaries

First, download the file lab2.py[4] into your `labs` folder (or folder of your choice).

Open the file in your Python editor and start up a Python interpreter.

If you haven't installed Anaconda, make sure you have installed NLTK (the Natural Language Toolkit) to work with your installation of Python.

Check that nltk is working by downloading the corpus for this lab:

```
import nltk
nltk.download('dependency_treebank')
```

[1]http://homepages.inf.ed.ac.uk/sgwater/teaching/lsa2015/labs/lab2.html

[2]http://homepages.inf.ed.ac.uk/sgwater/teaching/lsa2015/labs/lab2.pdf

[3]http://www.nltk.org/

[4]http://homepages.inf.ed.ac.uk/sgwater/teaching/lsa2015/labs/lab2.py

[5]http://www.comp.leeds.ac.uk/amalgam/tagsets/upenn.html

[6]https://docs.python.org/3.4/tutorial/datastructures.html#tuples-and-sequences

Examining the POS Tagset

A POS tagset is the set of Part-of-Speech tags used for annotating a particular corpus. The Penn Tagset[5] is one such tagset which is widely used for English. Click on the link and have a look at the tagset.

Q1. Based on your intuition, guess the most and least frequent tags in a data. What is the difference between the tags DT and PDT? Can you distinguish singular and plural nouns using this tagset? If so, how? How many different tags are available for main verbs and what are they?

Running the code

You can run this week's lab in the iPython console of Spyder (or another iPython console) as follows:

```
%run lab2.py
```

If you get an error that Spyder can't find lab2.py, you may need to use the full pathname to that file. For example, if your lab2.py file is in C:/My Documents/CompLingLabs/, then type `%run "C:/My Documents/CompLingLabs/lab2`
Or if you get sick of that, then type `cd C:/My Documents/CompLingLabs/` to tell Spyder to "cd" (*change directory*) into the directory (folder) where your lab2.py file is, and after that you can use just `%run lab2.py`.

It should print the first tagged sentence, and first (word, tag) pair. It also should print basic statistics about the data like the total number of sentences and the average sentence length (number of words per sentence). Finally, it should produce a figure showing a histogram of the different sentence lengths in the data set. (To display the figure in a separate window, see Note 2 below.)

For subsequent runs, you can skip the printing by doing:

```
%run lab2.py -q
```

Note 1: If you use the green 'Run' arrow in Spyder instead of the `%run` command, you may not get all the information printing out. I'm not sure why that is!

Note 2: As you're doing the lab, when you change anything in `run.py`, make sure to *save* the file before running again, or the changes you made will not take effect.

Note 3: If you're using the iPython console in Spyder, it will by default display figures in the console rather than in a separate window. You may want to display figures in a separate window, which allows you to resize them to make the text more visible. To do so, go to the " Preferences" menu (in Windows, it's under `Tools`, in Mac it is under `python` at top left). Choose `IPython console` and click on the `Graphics` tab. Under `Graphics backend` select `Automatic` and click OK. To have this change take effect, you need to start a *new* iPython console by clicking on `Consoles -> Open an IPython console`. Running the code should now produce a figure in a separate window (although for me the window opens behind the Spyder window, which is not ideal.)

Understanding the tsents data structure

For this lab, we consider a small part of the Penn Treebank POS annotated data. This data consists of around 3900 sentences, where each word is annotated with its POS tag using the Penn POS tagset. To access the data, our code first imports the `dependency_treebank` from nltk.corpus package using the command `from nltk.corpus import dependency_treebank`. We then extract the tagged sentences using the following command (on line 95):

```
tsents = dependency_treebank.tagged_sents()
```

`tsents` contains a list of `tagged sentences`. A `tagged sentence` is a list of pairs, where each pair consists of a word and its POS tag. A pair is just a Tuple with two members, and a `Tuple` is a data structure that is similar to a list, except that you can't change its length or its contents. The Python Tuple documentation (for *Python 2_* or Python 3[6]) provides a useful summary introduction to tuples.

Once you've loaded lab2.py, `tsents[0]` contains the first tagged sentence. `tsents[0][0]` gives the first tuple in the first sentence, which is a (word, tag) pair, and `tsents[0][0][0]` gives you the word from that pair, `tsents[0][0][1]` its tag.

Computing the distribution of tags

Construct a frequency distribution of POS tags by completing the code in the `tag_distribution` function, which returns a dictionary with POS tags as keys and the number of word tokens with that tag as values. *Hint*: look at the `sent_length_distribution` function if you aren't sure what to do here.

Using `plot_histogram`, plot a histogram of the tag distribution with tags on the x-axis and their counts on the y-axis, ordered by descending frequency. *Hint*: To sort the items (i.e., key-value pairs) in a dictionary by their values, you can use:

```
sorted(mydict.items(), key=lambda x: x[1])
```

Providing `reverse=True` as the third argument gives the result in reverse order. `help(sorted)` is your friend.

Using the histogram or commands to the interpreter, answer the following questions:

Q2. How many distinct tags are present in the data? What are the 5 most frequent and least frequent tags in this data? How does this compare with your intuition from Q1?

Q3. What kind of distribution do you see in the histogram, and how does it compare to the histogram of sentence lengths?

Computing the conditional distribution of tags for each word

Construct a conditional frequency distribution (CFD) by completing the code in the `word_tag_distribution` function. A CFD is a dictionary whose values are themselves distributions, keyed by context or condition. In our case we want words as conditions == keys, with values a frequency distribution of tags *for that word*.

For example, for the word *book*, the value of your CFD should be a frequency distribution of the POS tags that occur with *book*. Uncomment the code at the bottom of the file once you've implemented the function to see the tags for *book*.

Q4. How many entries are there in your big CFD? Using your CFD, compute the level of *tag ambiguity* in the corpus. That is, on average, how many different tags does each word have? If you had a larger tagged corpus, would you expect the amount of tag ambiguity to be greater than, less than, or the same as in the current corpus, and why? (You might be able to think of arguments on both sides; if so please explain.)

Q5. What tags are used for the word *the*, and which is the most frequent tag? Does what you see make you wonder about the tagging process, and how it was checked?

Feel free to look at the distributions for other words too, like *in*, *book*, *eat*.

Now, let's look at which word(s) out of the whole corpus have the greatest number of distinct tags. To do so, use the following line of code:

```
pprint(sorted(word_tag_dist.items(), key=lambda x: len(x[1]), reverse=True)[:10])
```

(`pprint` (think *pretty print*) is often useful if you have lots of nested dictionaries, lists and tuples to look at. . .)

Q6: For the first word in this list, can you think of example sentences that use this word in a way that's consistent with each of its different tags?

Building a Unigram Tagger

We shall now build a simple POS tagger called a *unigram* tagger using the function `unigram_tagger`. This function takes three arguments:

- The first argument is a conditional frequency distribution, which can be generated using the `word_tag_distriution` you completed above.
- The second argument is the most frequent POS tag in the corpus.
- The third argument is a sentence that needs to be tagged.

The goal of this function is to tag the sentence using probabilities from the CFD and most frequent POS tag. In order to make it work, you must complete its helper function, called `ut1`, to process a single word. If the word is seen (present in the CFD), `ut1` should assign the most frequent tag for that word. For unseen words (not present in the CFD), it should assign the overall most frequent POS tag, as passed in as the 2nd argument.

Q7. Why is this called a Unigram tagger? How does it differ from an HMM tagger?

Q8. Run this simple tagger and tag the sentences a) "book a flight to London" and b) "I bought a new book". Look at the POS tags. Are there any errors in the POS tags? What caused the error(s), and do you think an HMM tagger would make the same error(s)?

NLTK Frequency Distribution functions (optional)

It's useful to practice writing your own code for things to make sure you understand how they work, but it's also useful to know about existing tools in NLTK. This section provides some example code demonstrating how to use NLTK's `FreqDist()` and `ConditionalFreqDist()` functions, which compute frequency distributions and conditional frequency distributions, respectively. These functions are similar to the ones we just implemented (`tag_distribution` and `word_tag_distribution`), but have several additional features, such as `tabulate()`, which prints a frequency distribution in tabular form and `plot()`, which plots it in inverse frequency order.

For input to these functions, we need a single long list of word and tag pairs. Use `dependency_treebank.tagged_words` to get this list from our corpus (you can paste this line into the interpreter, or add it to your `lab2.py` file and rerun that file):

```
wtPairs = dependency_treebank.tagged_words()
```

We then use a comprehension to extract just the list of words from wtPairs, and construct a FreqDist from that:

```
wfd=FreqDist(w for (w,t) in wtPairs)
wfd
wfd.plot(50)
```

And finally using `ConditionalFreqDist()`, we compute the conditional frequency distribution of word and tag pairs for our corpus like this:

```
cfd = ConditionalFreqDist(wtPairs)
the_fd = cfd['the']
the_fd
len(the_fd)
the_fd.tabulate()
```

`cfd['the']` gives the frequency distribution of tags for the word *the*. `tabulate()` prints this distribution in tabular form. Compare the result with the conditional frequency distribution computed in the previous section for the word *the*. Both should give the same results.

You can also reimplement `ut1` to take advantage of nltk. How might you do it? *Hint*: try `help(FreqDist)` to find a method you could use.

Going further (for the ambitious)

1. Suppose you wanted to develop an probabilistic POS tagger (say, an HMM tagger) for social media text such as Tweets. What are the steps you would need to go through in order to build the tagger? Do you think the Penn tagset would work well for Twitter data? What are some of the differences you might need to account for and how might you do it? Aside from the tag set, what other issues might you face in developing the tagger?

2. NLTK has libraries to train different taggers. Using these libraries, build unigram and Hidden Markov Model taggers and evaluate them. First, split the data into two parts(90%, 10%). Consider the first 90% of the data as training data and the remaining 10% of the data as testing data. Build taggers using the training data. Then run the taggers on the test data and evaluate the performance of the tagger.

`help(nltk.tag.hmm.HiddenMarkovModelTagger)` and `help(nltk.UnigramTagger)` are your friends. Note particularly the `evaluate` function for evaluating the tagging of a collection of test sentences.

Find some examples where the tagger made an error, and try to identify why. What are some ways you could try to improve the tagger, hopefully reducing these errors? Consider both changing the *model* and changing the *data*.