Practical 1: POS tagging and smoothing

This practical is worth 60 % of the coursework credits for this module. Its due date is Friday 9th of March 2018, at 21:00. The usual penalties for lateness apply, namely Scheme B, 1 mark per 8 hour period or part thereof.

The purpose of this assignment is to make you familiar with the application of finite-state technology and parameter estimation. In particular, we will be looking at part-of-speech (POS) tagging.

Getting started

First, you may want to open a Python interpreter, and enter the below lines one by one, and see what happens. Illustrated is how to access the tagged Brown corpus in NLTK. Further explanation is provided in the tutorials.

```
from nltk.corpus import brown

sents = brown.tagged_sents()
first = sents[0]
first
words = [w for (w,_) in first]
words
tags = [t for (_,t) in first]
tags

def show_sent(sent):
    print(sent)

for sent in sents[0:10]:
    show_sent(sent)
```

Especially for inexperienced Python programmers, it may be worthwhile to try things out in the interpreter. For further developing your code however,

it is preferable to write a Python program in a file and call that from the (Linux) command line.

Explore different tag sets

Notice that the raw tags from the Brown corpus sometimes consist of two parts separated by a hyphen. You may choose to remove the second parts for the purpose of the experiments later.

Also try:

```
from nltk.corpus import brown

sents = brown.tagged_sents(tagset='universal')
first = sents[0]
first
```

Do you see the difference?

Now find some more tagged corpora in NLTK, possibly some for different languages, for use in the experiments we will be doing later. For example:

```
from nltk.corpus import conll2000, alpino, floresta
sents_en = conll2000.tagged_sents()
sents_enu = conll2000.tagged_sents(tagset='universal')
sents_nl = alpino.tagged_sents()
sents_po = floresta.tagged_sents()
```

Note that some of these corpora have very different types of tags, and for some tag sets it may be appropriate to prune the names, as in the case of the hyphenated names in the Brown corpus. Further note that some tag sets are larger than others; POS tagging is generally more difficult for larger tag sets than for smaller ones, which you may be able to observe in the experiments to be discussed next.

POS tagging

We will now be developing a first-order HMM (Hidden Markov Model) for POS (part of speech) tagging in Python. First, we write code to estimate the transition probabilities and the emission probabilities of an HMM, on the basis of (tagged) training sentences from a corpus. Do not forget to involve the start-of-sentence and end-of-sentence markers in the estimation.

Second, we write code to apply a trained HMM on each (untagged) sentence from the testing part of a corpus to determine the sequence of tags that

has the highest probability. Third, we write code to compare that sequence with the 'gold-standard' sequence of tags for that sentence, i.e. we determine the percentage of tags in the test corpus that is guessed correctly.

As training set, we may take e.g. the first 10,000 sentences from the Brown corpus, and as test set, we may take a further 500 sentences from the same corpus (not overlapping with the training set). **(Avoid iterating over every possible POS for every word, or tagging 500 sentences will take a long time! For each word, consider only the POSs that have been seen for that word during training.)**

You will need to ensure the HMM can handle all input. The two main concerns are unknown words and smoothing:

- In order to deal with words in the test sentences that do not occur in the training sentences, implement code to estimate the probability of an unknown word, given a tag. A simple way of doing this is to look for words that occur exactly once in the training sentences, and to replace them by an artificial 'UNK' (unknown) word during training of the HMM. During testing, we then replace unknown words (that is, words not occurring (more than once) in the training sentences) by UNK before we compute the most likely sequence of tags. In this way, (almost) all emission probabilities will be non-zero.

- Smooth the transition probabilities, to make the transition between each pair of tags be non-zero. Use Laplace smoothing in the first instance.

Experiments

Do experiments with some of the available corpora in NLTK (a minimum of five). Measure overall accuracy, and determine for which parts of speech the accuracy is lowest and for which it is highest. Try to explain what you measure.

Extensions

The first proposed extension is to implement from scratch, and from your own understanding, one or more of:

- Good-Turing smoothing (this will likely require you to read up on simple linear regression)

- Katz backoff (typically requires Good-Turing smoothing, but could possibly be done with Laplace smoothing instead)
- Stupid backoff
- Kneser-Ney smoothing
- Interpolation

Be careful to do this from your own understanding and don't peek at the implementations in NLTK or from elsewhere. If you do base your implementation on an existing implementation, acknowledge this and spell out how much of the submitted code is your own contribution.

It is generally better to have a simple implementation than a complicated one that the marker cannot understand. Also, this should not be a contest to get the best accuracy; gaining good understanding of the concepts is more important.

You may also read up on smoothing techniques other than those discussed in the lectures, explain them in your report, and, where realistic, implement them, and compare accuracy with other techniques.

Another extension could be to experiment with a more general approach to handling unknown words. Instead of having a single UNK pseudo-word, you could have several, depending on the first or last few letters of an input word. For example, in English a word ending on -ing is likely a present participle or gerund, and a capitalised word is likely a proper noun (unless it occurs at the start of a sentence). So one could have UNK-ing, UNK-cap, etc., used in much the same way as before during both training and testing.

Limited extra credit is given for exploring SimpleGoodTuringProbDist, WittenBellProbDist or KneserNeyProbDist as implemented in NLTK. How well do they work? What do their arguments mean? (For both SimpleGoodTuringProbDist and WittenBellProbDist you may find that the `bins` argument is needed for proper functioning.)

Requirements

For up to 14 marks, the requirements of this practical are:

- Implement training of the HMM using relative frequency estimation, with handling of unknown words and Laplace smoothing.
- Implement computation of the most probable sequence of tags for a given (untagged) sentence, according to the trained HMM.

- Implement code to measure accuracy of the tagged sentence.
- Implement code that runs the training, the tagging and the measurement of accuracy, for at least five corpora.
- Add a plain-text 'readme' file that tells me how to run this code from the command line (no IDEs).

For up to 17 marks, further required is the following:

- Write a report of about 2 pages in which you describe how you have designed the training and testing of the HMM, and what accuracies you have found for the various corpora and tag sets, and with which smoothing techniques. For which tags did you get the best and worst accuracy? (Confusion matrices might be helpful.) Any other observations about how accuracies differ between languages, and between tag sets of different sizes?

For more than 17 marks, you need to have done at least one extension, and have described this in detail in your report.

Hints

Even though this module is not about programming per se, a good programming style is expected. Choose meaningful variable and function names. Break up your code into small functions. Avoid cryptic code, and add code commenting where it is necessary for the reader to understand what is going on. Do not overengineer your code; a relatively simple task deserves a relatively simple implementation.

You cannot use any of the POS taggers already implemented in NLTK. However, you may use general utility functions in NLTK such as `ngrams` from `nltk.util`. Also `FreqDist` from `nltk` could be useful.

On the lab machines (Fedora), it is best to use Python 2, as NLTK is not provided for Python 3. If you have strong reasons to use Python 3, then the appendix tells you how. Either way, make sure that your code runs on the lab machines and that you provide me with all necessary instructions to test your code.

When you are reporting the outcome of experiments, the foremost requirement is reproducibility. So if you give figures or graphs in your report, explain precisely what you did, and how, to obtain those results.

Pointers

- Marking
  `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`
- Lateness
  `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`
- Good Academic Practice
  `https://www.st-andrews.ac.uk/students/rules/academicpractice/`

A NLTK and Python 3

If you want to use NLTK in Python 3 on the Fedora lab machines, then download `nltksetup.sh` from this directory, put it into the directory with your Python code, then run once:

```
sh nltksetup.sh
```

Thereafter, whenever you resume working on your assignment and want to run your code, do first:

```
. ./nltkvenvCS5012/bin/activate
```

When you are done for the day, you can run:

```
deactivate
```

A simpler suggestion is to use Python 2. But note that there are annoying differences between Python 2 and Python 3. For one thing, division of integers in Python 2 returns an integer result. So if you want to divide integer `n` by integer `m`, then you may want to do `1.0 * n / m` rather than `n / m` if you expect a floating-point number to be returned; the `1.0 * n` turns `n` into a floating-point number, with the desired result.