Practical 1: POS tagging and unknown words

This practical is worth 60% of the coursework credits for this module (24% of the final grade). Its due date is Tuesday 10th of March 2020, 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 gain understanding of the Viterbi algorithm, and its application to part-of-speech (POS) tagging.

Getting started

For this practical we use Python3. On the lab (Linux) machines, you need the full path `/usr/local/python/bin/python3`, which is set up to work with NLTK. (Plain `python3` won't be able to find NLTK.)

If you run `python` on your personal laptop, be sure it is Python3 rather than Python2; in case of doubt, do `python --version`.

To get started with the tagged corpora, 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(tagset='universal')
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.

Viterbi algorithm

You will develop a first-order HMM (Hidden Markov Model) for POS (part of speech) tagging in Python. This involves:

- counting occurrences of one part of speech following another in a training corpus,
- counting occurrences of words together with parts of speech in a training corpus,
- relative frequency estimation with smoothing,
- finding the best sequence of parts of speech for a list of words in the test corpus, according to a HMM model with smoothed probabilities,
- computing the accuracy, that is, the percentage of parts of speech that is guessed correctly.

As discussed in the lectures, smoothing is necessary to avoid zero probabilities for events that were not witnessed in the training corpus. Rather than implementing a form of smoothing yourself, you can for this assignment take the implementation of Witten-Bell smoothing in NLTK (among the forms of smoothing in NLTK, this seems to be the most robust one). An example of use for emission probabilities is in file `smoothing.py`; one can similarly apply smoothing to transition probabilities.

Run your application on the Brown corpus (with the universal tagset). If you use the first 10,000 sentences for training and the next 500 sentences for testing, then the accuracy should be around 95%. If your accuracy is much lower, then you are probably doing something wrong.

Unknown words

Any words that were not seen in the training corpus are exchangeable as far as the Viterbi algorithm is concerned. But there is a lot of information in the spelling of unknown words that could help us guess their parts of speech. For example, if a word

is not the first word in a sentence and it is capitalised, then it is likely a proper noun. If a word ends on '-ing', then it is likely a verb or noun (gerund).

This observation motivates a phase of preprocessing of the tagged sentences, preceding both training and testing. For training, we first determine which words are infrequent in the training corpus. Infrequent could mean occurring only once (or perhaps no more than twice, ...; there is a degree of freedom here). Then, if a word is infrequent and has a certain pattern, for example it ends on '-ing', then we replace the word by a special tag, say 'UNK-ing'. Training then proceeds as usual. For testing, we do something similar. If a word did not occur in the training corpus or only infrequently, and has a certain pattern, then it is replaced by the appropriate tag. Testing then proceeds as usual.

Do experiments to determine whether use of UNK tags for capitalisation and the '-ing' suffix helps to improve accuracy. Investigate whether it helps to introduce more UNK tags.

Other languages

The UNK-ing tag is specific to English. Consider one or more other languages. What would be appropriate UNK tags? You may want to read up a little on the morphology of those languages, and/or investigate the predictive power of prefixes and suffixes to determine the part of speech, by investigating the corpora in NLTK.

Corpora for several other languages are available in the same package `nltk.corpus`, but regrettably without the universal tagset (you need to remove the `tagset='universal'` argument, say for floresta or alpino). The larger tagsets will slow down testing considerably. To avoid this, it is best to simplify composite tags like `JJT-HL` by just their first part, here `JJT`. It is advisable to read up on what these tags mean.

Requirements

Submit the Python code, with a README file telling me how to run the code, and a report describing your findings, including experimental results and their analysis.

Marking is in line with the General Mark Descriptors (see pointers below). Evidence of an acceptable attempt (up to 7 marks) could be code that is not functional but nonetheless demonstrates some understanding of the Viterbi algorithm. Evidence of a competent attempt addressing most requirements (up to 13 marks) could be fully correct code in good style, but without investigation of UNK tags or without considering other languages.

Evidence of exceptional achievements (19-20 marks) would be thorough investigation of UNK tags and convincing documentation whether and how they help to improve accuracy, for English and several other languages, with discussion of morphology and tagsets. This would typically involve informative graphs, tables, and so on, in the 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. If you feel the need for Python virtual environments, then you are probably overdoing it. The code that you upload would typically consist of one or two `.py` files.

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`, and `FreqDist` and `WittenBellProbDist` from `nltk`.

There is a chance that you will experience underflow when running the Viterbi algorithm. For this reason, you may omit consideration of sentences longer than 100 tokens.

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/`