Practical 1: Parts of speech in the Universal Dependencies treebanks

This practical is worth 50% of the coursework credits for this module. Its due date is Friday 11th of March 2022, 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. The Viterbi algorithm will be related to two other algorithms.

You will also get to see the Universal Dependencies treebanks. The main purpose of these treebanks is dependency parsing (to be discussed later in the module), but here we only use their part-of-speech tags.

Getting started

We will be using 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`. Next to NLTK (`https://www.nltk.org/`), you will also need to install the `conllu` package (`https://pypi.org/project/conllu/`).

To help you get started, download `gettingstarted.py` and the other Python files, and the zip file with treebanks from this directory. After unzipping, run `/usr/local/python/bin/python3 gettingstarted.py`. You may, but need not, use parts of the provided code in your submission.

The three treebanks come from Universal Dependencies. You can download the entire set of treebanks from `https://universaldependencies.org/`.

Parameter estimation

First, we write code to estimate the transition probabilities and the emission probabilities of an HMM (Hidden Markov Model), on the basis of (tagged) sentences from a training corpus from Universal Dependencies. Do not forget to involve the start-of-sentence marker $\langle s \rangle$ and the end-of-sentence marker $\langle /s \rangle$ in the estimation.

The code in this part is concerned with:

- 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.

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 implementations 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.

Three algorithms for POS tagging

Algorithm 1: eager algorithm

First, we implement a naive algorithm that chooses the POS tag for the i-th token on the basis of the chosen $(i-1)$-th tag and the i-th token. To be more precise, we determine for each $i = 1, \ldots, n$, in this order:

$$\hat{t}_i = \operatorname*{argmax}_{t_i} P(t_i \mid \hat{t}_{i-1}) \cdot P(w_i \mid t_i)$$

assuming $\hat{t}_0$ is the start-of-sentence marker $\langle s \rangle$. Note that the end-of-sentence marker $\langle /s \rangle$ is not even used here.

Algorithm 2: Viterbi algorithm

Now we implement the Viterbi algorithm, which determines the sequence of tags for a given sentence that has the highest probability. As discussed in the lectures, this is:

$$\hat{t}_1 \cdots \hat{t}_n = \operatorname*{argmax}_{t_1 \cdots t_n} \left(\prod_{i=1}^{n} P(t_i \mid t_{i-1}) \cdot P(w_i \mid t_i) \right) \cdot P(t_{n+1} \mid t_n)$$

where the tokens of the input sentence are $w_1 \cdots w_n$, and $t_0 = \langle s \rangle$ and $t_{n+1} = \langle /s \rangle$ are the start-of-sentence and end-of-sentence markers, respectively.

To avoid underflow for long sentences, we need to use log probabilities.

Algorithm 3: individually most probable tags

We now write code that determines the most probable part of speech for each token individually. That is, for each i, computed is:

$$\hat{t}_i = \operatorname*{argmax}_{t_i} \sum_{t_1 \cdots t_{i-1} t_{i+1} \cdots t_n} \left(\prod_{i=1}^{n} P(t_i \mid t_{i-1}) \cdot P(w_i \mid t_i) \right) \cdot P(t_{n+1} \mid t_n)$$

To compute this effectively, we need to use forward and backward values, as discussed in the lectures on the Baum-Welch algorithm, making use of the fact that the above is equivalent to:

$$\begin{aligned} \hat{t}_i = \operatorname*{argmax}_{t_i} \; & \textstyle\sum_{t_1 \cdots t_{i-1}} \Big(\prod_{k=1}^{i} P(t_k \mid t_{k-1}) \cdot P(w_k \mid t_k) \Big) \cdot \\ & \textstyle\sum_{t_{i+1} \cdots t_n} \Big(\prod_{k=i+1}^{n} P(t_k \mid t_{k-1}) \cdot P(w_k \mid t_k) \Big) \cdot P(t_{n+1} \mid t_n) \end{aligned}$$

The computation of forward values is very similar to the Viterbi algorithm, so you may want to copy and change the code you already had, replacing statements that maximise by corresponding statements that sum values together. Computation of backward values is similar to computation of forward values.

See `logsumexptrick.py` for a demonstration of the use of log probabilities when probabilities are summed, without getting underflow in the conversion from log probabilities to probabilities and back.

Evaluation

Next, we write code to determine the percentages of tags in a test corpus that are guessed correctly by the above three algorithms. Run experiments for the training and test corpora of the three included treebanks, and possibly for treebanks of more languages (but not for more than 10; aim for quality rather than quantity). Compare the performance of the three algorithms.

You get the best experience out of this practical if you also consider the languages of the treebanks. What do you know (or what can you find out) about the morphological and syntactic properties of these languages? Can you explain why POS tagging is more difficult for some languages than for others?

Requirements

Submit your Python code, with a README file explaining how to run it, and a report describing your findings, including experimental results and their analysis. You would include the treebanks needed to run the code, **but please do not include the entire set of Universal Dependencies treebanks, because this would be a huge waste of disk space and band width for the marker**.

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 POS tagging. Evidence of a reasonable attempt (up to 10 marks) could be code that implements Algorithm 1. Evidence of a competent attempt addressing most requirements (up to 13 marks) could be fully correct code in good style, implementing Algorithms 1 and 2 and a brief report. Evidence of a good attempt meeting nearly all requirements (up to 16 marks) could be a good implementation of Algorithms 1 and 2, plus an informative report discussing meaningful experiments. Evidence of an excellent attempt with no significant defects (up to 18 marks) requires an excellent implementation of all three algorithms, and a report that discusses thorough experiments and analysis of inherent properties of the algorithms, as well as awareness of linguistic background discussed in the lectures. An exceptional achievement (up to 20 marks) in addition requires exceptional understanding of the subject matter, evidenced by experiments, their analysis and reflection 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 at most a handful of `.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`.

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. Keep in mind that not all treebanks are the same size, and a common pitfall is making unfair comparisons between languages by not correcting for differences in corpus size.

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/`