Practical 1: Part of speech tagging and unknown words

This practical is worth 50% of the coursework component of this module. Its due date is Friday 10th of March 2023, at 21:00. Note that MMS is the definitive source for deadlines and weights.

The purpose of this assignment is to gain understanding of the Viterbi algorithm, and its application to part-of-speech (POS) tagging.

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. If you are interested, 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.

Viterbi algorithm

On the basis of the estimated parameters, you can implement the Viterbi algorithm, to determine the best sequence of parts of speech for a list of words from the test corpus, according to an HMM model with smoothed probabilities. You can then determine the accuracy, that is, the percentage of parts of speech that is guessed correctly, over the whole corpus. For the English corpus, you should now be able to get an accuracy above 90%. If you get less than 88% for this corpus, it is likely you did not (correctly) implement the Viterbi algorithm. The accuracy for the Ukrainian corpus is going to be lower than for the English corpus.

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 for the English corpus 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 (up to 10, but no more). One of these UNK tags may be a 'catch-all', which stands for any infrequently occurring word other than those ending on '-ing' or capitalised, etc.

Now turn to the French and Ukrainian corpora. You can start by introducing a single UNK tag to capture all infrequently occurring words. Does this already help to improve accuracy? Next you can introduce more UNK tags for certain suffixes (or even prefixes), up to 10 for each language. This is understandably more difficult than for English, unless you happen to speak these languages. However, by exploring the corpora, or by doing some additional reading, you might guess suitable affixes.

Requirements

Submit your Python code and the report.

It should be possible to run your implementation of the Viterbi algorithm on one or all three of the corpora simply by calling from the command line:

```
python3 p1.py
```

You may add further functionality, but then add a README file to explain how to run that functionality. You would include the three treebanks needed to run the code, **but please do not include the entire set of 244 treebanks from Universal Dependencies, because this would be a huge waste of disk space and band width for the marker**.

In the report, you write up your findings. What were the accuracies for the three corpora before introducing UNK tags? How much did the accuracies improve thanks to the UNK tags? How did you choose the UNK tags? Can you say anything about these three corpora that could explain why different accuracies were obtained? I.e. can a language or corpus have properties that make part-of-speech tagging become easier or more difficult? An appropriate length of a report could be 2 or 3 pages of text.

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 functional code that implements some algorithm for POS tagging, but not the Viterbi algorithm. Evidence of a competent attempt addressing most requirements (up to 13 marks) could be fully correct code in good style, implementing the Viterbi algorithm, and a brief report. Evidence of a good attempt meeting nearly all requirements (up to 16 marks) could be a good implementation of the Viterbi algorithm, plus an informative report discussing some choice of UNK tags for at least English. Evidence of an excellent attempt with no significant defects (up to 18 marks) requires an excellent implementation of the Viterbi algorithm, and an excellent report discussing and motivating UNK tags for all three languages, with awareness of the linguistic properties of these languages. 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.

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.

Considering current class sizes, please be kind to your marker, by making their task as smooth as possible:

- Go for quality rather than quantity. We are looking for evidence of understanding rather than for lots of busywork. Especially understanding of language and how language works from the perpective of the HMM model is what this practical should be about.

- Avoid Python virtual environments. These blow up the size of the files that markers need to download. If you feel the need for Python virtual environments, then you are probably overdoing it, and mistake this practical for a software engineering project, which it most definitely is not. The code that you upload would typically consist of two or three `.py` files.

- You could use standard packages such as `numpy` or `pandas`, which the marker will likely have installed already, but avoid anything more exotic. Assume a version of Python3 that is the one on the lab machines or older; the marker may not have installed the latest bleeding-edge version yet.

- We strongly advise against letting the report exceed 10 pages. We do **not** expect an essay on NLP or the history of the Viterbi algorithm, or anything of the sort.

- It is fine to include a couple of graphs and tables in the report, but don't overdo it. Plotting accuracy against any conceivable hyperparameter, for the sake of producing plots, is **not** what we are after.

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/`