Practical 1: (Small) language models

This practical is worth 50% of the coursework component of this module. Its due date is Monday 17th of March 2025, 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, and of language modelling using HMMs (Hidden Markov Models) and bigrams.

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 and NLTK, introduced in the Week 2 tutorial. Also needed is the Python package `conllu` (typically installed using pip).

To help you get started, this directory provides three Python files as well as a zip file with treebanks. After unzipping the treebanks, run `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/`, but only include the three treebanks provided here with your final submission.

HMMs

Write your own code to estimate the transition probabilities and the emission probabilities of an HMM, 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. For emission probabilities, implement Witten-Bell smoothing for unigrams, exactly as on slide 23 of Lecture 5 (use 100000 as value of z). This means that we have one smoothed probability distribution over words for each tag. For transition probabilities, implement Witten-Bell smoothing for bigrams of tags, exactly as on slide 22 of that same lecture (and this in turn will make use of Witten-Bell unigram smoothing that you will have implemented before). To be clear: do **not** use any existing implementations of Witten-Bell, but make your own, based on your understanding of the lecture notes.

Further write your own code to implement the Viterbi algorithm, which determines the sequence of tags for a given sentence that has the highest probability. To avoid underflow for long sentences, we need to use log probabilities. Implement code to compute accuracy of POS tagging (the percentage of correctly predicted tags).

Language modelling using HMMs

As discussed in the lectures, HMMs can be used to determine the probability of an input sentence, using the forward algorithm. For this, you need to be able to add probabilities together that are represented as log probabilities, without getting underflow in the conversion from log probabilities to probabilities and back. See the included `logsumexptrick.py` for a demonstration.

Write your own code to compute the perplexity of a test corpus. (Consider the length of a corpus to be the total number of actual tokens plus the number of sentences; in effect we count one additional end-of-sentence token for each sentence in addition to the actual words and punctuation tokens.)

Language modelling using bigrams

Further write your own code to implement estimation of bigram probabilities of input tokens (so words and punctuation tokens, not POS tags). To avoid zero probabilities, you again need Witten-Bell smoothing; you should here be able to reuse the code you implemented earlier for transition probabilities of HMMs.

Again, implement your own code to compute perplexity of a test corpus, given a trained bigram model.

Experiments

Run the developed code for the three treebanks. Train using the training parts of the treebanks, and test using the testing parts of the treebanks. (It is good practice to mainly use the development parts during development of the code.) Testing here means:

- computing accuracy of POS tagging using an HMM,
- computing perplexity using an HMM, and
- computing perplexity using a bigram model.

Requirements

Submit your Python code and the report.

It should be possible to run your code simply by calling from the command line:

```
python3 p1.py
```

This should print to standard output (no fancy graphics) the accuracy and the perplexities, for each of the three corpora. For this to work, you of course need to include the three provided treebanks. Do **not** submit Jupyter Notebooks please.

The report should consist of the following sections:

Implementation (at most 1 page): notes on your implementation of HMMs and bigrams.

Results (at most $\frac{1}{2}$ page): the outcomes of the experiments, in terms of accuracies and perplexities, ideally represented in the form of one or two tables.

Reflection (at most $1\frac{1}{2}$ page): reflection on the outcomes of the experiments.

The above length limits are advisory. This means it is not recommended to submit any more text than indicated. Exceeding the limits by a small proportion would not normally be penalised, but no credit will be given for large amounts of text that this specification does not ask for, and being unnecessarily verbose can affect the mark in a negative way.

Marking guidance

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 or language modelling. Evidence of a reasonable attempt (up to 10 marks) could be code for the Viterbi algorithm that is functional but produces incorrect experimental results. Evidence of a competent attempt addressing most requirements (up to 13 marks) could be fully correct code for the Viterbi algorithm, in good style, and a brief report, but nothing else. Evidence of a good attempt meeting nearly all requirements (up to 16 marks) could be a good implementation of all required algorithms, plus an informative report. Evidence of an excellent attempt with no significant defects (up to 18 marks) requires an excellent implementation of all algorithms, and a report that shows excellent insight into the algorithms and of linguistic background discussed in the lectures. An exceptional achievement (up to 20 marks) in addition requires a particularly elegant (not large) implementation and exceptional understanding of the subject matter evidenced by deep 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 may not use any of the POS taggers already implemented in NLTK. However, you may use `bigrams` and `FreqDist` from `nltk.util`.

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

- Do not submit Python virtual environments. These blow up the sizes of the files that markers need to download. If you feel the need for exotic libraries, 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 three or four short `.py` files.

- 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.

Policies

- See the Generic Mark Descriptors in the School Student Handbook
`http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`
- The standard penalty for late submission applies (Scheme A: 1 mark per 24 hour period, or part thereof)
`http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`
- The University policy on Good Academic Practice applies
`https://www.st-andrews.ac.uk/students/rules/academicpractice/`