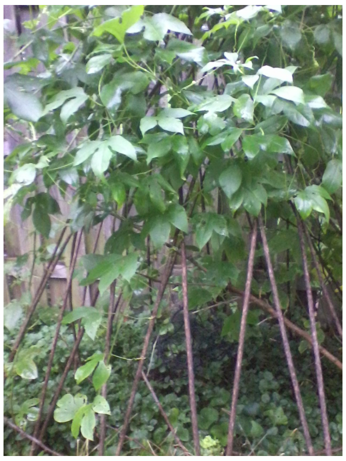

CS440/ECE448 Fall 2023

MP 8: HMM POS tagging

Due date: Monday October 30th, 11:59pm

For this MP, improve your (POS) tagging model from MP7. Make sure you understand the algorithm before you start writing code, e.g. look at the lectures on Hidden Markov Models and Chapter 8 of Jurafsky and Martin.

General guidelines

Basic instructions are the same as in previous MPs:

- For general instructions, see the main MP page and the course policies.
- You may use numpy (though it's not needed). You may not use other non-standard modules (including nltk).

Problem Statement

The MP8 code reads data from two files. Your tagging function will be given the training data with tags and the test data without tags. Your tagger should use the training data to estimate the probabilities it requires, and then use this model to infer tags for the test input. The main MP8 function will compare these against the correct tags and report your accuracy.

The data is divided into sentences. Your tagger should process each sentence independently.

You will need to write two additional modifications to the Viterbi tagger you wrote in MP7.

The materials

The code package (and data) for the MP (template.zip) contains these code files:

- mp8.py (do not change)
- utils.py (do not change)
- baseline.py
- viterbi
  - viterbi_2.py
  - viterbi_3.py
- test_viterbi
  - test_viterbi.py
  - utils.py (do not change)

The code package contains the following training and development data:

- Brown corpus:
  - data/brown-training.txt
  - data/brown-dev.txt
- Small synthetic testing dataset (for test_viterbi.py):
  - data/mttest-training.txt
  - data/mttest-dev.txt

You should use the provided training data to train the parameters of your model and the development sets to test its accuracy.

To run the code on the Brown corpus data you need to tell it where the data is and which algorithm to run, either baseline/viterbi_1 from MP7, or viterbi_2/viterbi_3:

```
python3 mp8.py --train data/brown-training.txt --test data/brown-dev.txt --algorithm [viterbi_2, viterbi_3]
```

The program will run the algorithm and report three accuracy numbers:

- overall accuracy
- accuracy on words that have been seen with multiple different tags
- accuracy on unseen words

Many words in our datasets have only one possible tag, so it's very hard to get the tag wrong! This means that even very simple algorithms have high overall accuracy. The other two accuracy numbers will help you see where there is room for improvement.

Code Submission

You will be submitting two files to Gradescope: **viterbi_2.py** and **viterbi_3.py**. Each of these submit points times out after 10 minutes. Bear in mind that your laptop may be faster than the processor on Gradescope.

The Gradescope autograder will run your code on the development dataset from the supplied template and also a separate (unseen) set of data from the same corpus. In addition, your code will be tested on one or more hidden datasets that are not available to you, which may have different number of tags and words from the ones provided to you.

Do NOT hardcode any of your important computations, such as transition probabilities and emission probabilities, number or name of tags, and etc.

Tagset

The following is an example set of 16 part of speech tags. This is the tagset used in the provided Brown corpus. **But remember you should not hardcode anything regarding this tagset because we will test your code on another dataset with a different tagset**.

- ADJ adjective
- ADV adverb
- IN preposition
- PART particle (e.g. after verb, looks like a preposition)
- PRON pronoun
- NUM number
- CONJ conjunction
- UH filler, exclamation
- TO infinitive
- VERB verb
- MODAL modal verb
- DET determiner
- NOUN noun
- PERIOD end of sentence punctuation
- PUNCT other punctuation
- X miscellaneous hard-to-classify items

The provided code converts all words to lowercase when it reads in each sentence. It also adds two dummy words at the ends of the sentnce: START (tag START) and END (tag END). These tags are just for standardization; they will not be considered in accuracy computations.

Since the first word in each input sentence is always START, the initial probabilities for your HMM can have a very simple form, which you can hardcode into your program. Your code only needs to learn the transition probabilities and the emission probabilities. The transition probabilities from START to the next tag will encode the probability of starting a sentence with the various different tags.

Viterbi_2

The previous Vitebi tagger fails to beat the baseline because it does very poorly on unseen words. It's assuming that all tags have similar probability for these words, but we know that a new word is much more likely to have the tag NOUN than (say) CONJ. For this part, you'll improve your emission smoothing to match the real probabilities for unseen words.

Words that occur only once in the training data ("hapax" words) have a distribution similar to the words that appear only in the test/development data. Extract these words from the training data and calculate the probability of each tag on them. When you do your Laplace smoothing of the emission probabilities for tag T, **scale Laplace smoothing constant α by the corresponding probability of tag T occurs among the set hapax words.**

This optimized version of the Viterbi code should have a significantly better unseen word accuracy on the Brown development dataset, e.g. over 66.5%. It also beat the baseline on overall accuracy (e.g. 95.5%). You should write optimized version of Viterbi as the viterbi_2 function in vertibi.py.

The hapax word tag probabilities may be different from one dataset to another. So your Viterbi code should compute them dynamically from its training data each time it runs.

Hints:

- Most of the code in viterbi_2.py is the same as that of viterbi_1.py
- Tag X rarely occurs in the dataset.
- Setting a high value for the Laplace smoothing constant may overly smooth the emission probabilities and break your statistical computations. A small value for the Laplace smoothing constant, e.g. 1e-5, may help.
- We do not recommend using global variables in your implementation since Gradescope runs a number of different tests within the same python environment. Global values set during one test will carry over to subsequent tests.

Viterbi_3

The task for this last part is to maximize the accuracy of the Viterbi code. You must train on only the provided training set (no external resources) and you should keep the basic Viterbi algorithm. However, you can make any algorithmic improvements you like. This optimized algorithm should be named viterbi_3.

We recommend trying to improve the algorithm's ability to guess the right tag for unseen words. If you examine the set of hapax words in the training data, you should notice that words with certain prefixes and certain suffixes typically have certain limited types of tags. For example, words with suffix "-ly" have several possible tags but the tag distribution is very different from that of the full set of hapax words. You can do a better job of handling these words by changing the emissions probabilities generated for them.

Recall what we did for Viterbi 1 and 2: we mapped hapax words (in the training data) and unseen words (in the development or test data) into a single pseudoword "UNKNOWN". To exploit the form of the word, you can map hapax/unseen words into several different pseudowords. E.g. perhaps all the words ending in "-ing" could be mapped to "X-ING". Then you can use the hapax words to calculate suitable probability values for X-ING, as you did for Viterbi_2.

Hints:

- When you map a new word to the set of "X-ING", you may find that changing the weight you add is useful. For example, if you start with

  ```
  emit_prob[tag][pattern] += 1
  ```

  you may want to change the increment:

  ```
  emit_prob[tag][pattern] += something_else
  ```

It is extremely hard to predict useful prefixes and suffixes from first principles. They may be useful patterns that aren't the kind of prefix or suffix you'd find in an English grammar, e.g. first names ending in -a are likely to be women's names in many European languages. We strongly recommend building yourself a separate python tool to dump the hapax words, with their tags, into a separate file that you can inspect. You may assume that our completely hidden dataset is in English, so that word patterns from the Brown corpus should continue to be useful for the hidden corpus.

Using this method, our model solution gets over 76% accuracy on unseen words, and over 96% accuracy overall. (Both numbers on the Brown development dataset.)

It may also be possible to improve performance by using two previous tags (rather than just one) to predict each tag. A full version of this idea would use 256 separate tag pairs and may be too slow to run on the autograder. However, you may be able to gain accuracy by using only selected information from the first of the two tags. Also, beam search can be helpful to speed up decoding time.