CS440/ECE448 Spring 2020

Assignment 4: HMM POS tagging

Due date: Wednesday March 25th, 11:59pm

Fall 2018: Margaret Fleck, Renxuan Wang, Tiantian Fang, Edward Huang
Spring 2020: Jialu Li, Guannan Guo, and Kiran Ramnath
adapted from a U. Penn assignment

For this MP, you will implement part of speech (POS) tagging using an HMM model Make sure you understand the algorithm before you start writing code, e.g. look at lectures 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.
- Extra credit is available, capped at 10%.
- You may use numpy (though it's not needed). You may not use other non-standard modules (including nltk).

Problem Statement

The mp4 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 to estimate the probabilities it requires, and then use this model to infer tags for the test input. The main mp4 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 tagging functions:

- Baseline
- Viterbi: HMM tagger

The materials

The code package for the MP (zip) contains three files:

- mp4.py
- utils.py
- viterbi.py
- extra.py

You will be submitting a modified version of viterbi.py and/or extra.py for extra credits, which must work with our provided versions of the other two files.

The code package also contains two sets of training and development data

- Brown corpus
- MASC corpus (from the Open American National Corpus)

The provided code converts all words to lowercase. You should test your code on these two datasets.

You should use the training data to train the parameters of your model and the development sets to test its accuracy. We will use a separate (unseen) set of data to test your code after you submit it.

In addition, your code will be test on a hidden dataset that is not available to you, which has different number of tags and words from the ones provided to you. **So do NOT hardcopy any of your important computations, such as initial probabilities, transition probabilities, emission probabilities, number or name of tags, and etc.**

Here is an example of how to run the code on the Brown corpus data:

```
python3 mp4.py --train data/brown-training.txt --test data/brown-dev.txt
```

The program will run both baseline and Viterbi algorithms, reporting three accuracy numbers:

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

Tagset

For this MP, we will use the following set of 16 part of speech tags **for the visible dataset that is available to you**:

- 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

Baseline tagger

The Baseline tagger considers each word independently, ignoring previous words and tags. For each word w, it counts how many times w occurs with each tag in the training data. When processing the test data, it consistently gives w the tag that was seen most often. For unseen words, it should guess the tag that's seen the most often in training dataset.

A correctly working baseline tagger should get about 93% accuracy on the Brown corpus development set.

DO NOT ATTEMPT TO IMPROVE THE BASELINE CODE.

Part 1 Viterbi

The Viterbi tagger should implement the HMM trellis (Viterbi) decoding algoirthm as seen in lecture or Jurafsky and Martin. That is, the probability of each tag depends only on the previous tag, and the probability of each word depends only on the corresponding tag. This model will need to estimate three sets of probabilities:

- Initial probabilities (How often does each tag occur at the start of a sentence?)
- Transition probabilities (How often does tag t_b follow tag t_a?)
- Emission probabilities (How often does tag t yield word w?)

It's helpful to think of your processing in five steps:

- Count occurrences of tags, tag pairs, tag/word pairs.
- Compute smoothed probabilities
- Take the log of each probability
- Construct the trellis. Notice that for each tag/time pair, you must store not only the probability of the best path but also a pointer to the previous tag/time pair in that path.
- Return the best path through the trellis.

You'll need to use smoothing to get good performance. Make sure that your code for computing the three types of probabilities never returns zero. Laplace smoothing is the method we use to smooth zero probability cases for calculating initial probabilities, transition probabilities, and emission probabilities.

For example, to smooth the emission probabilities, consider each tag individually. For a fixed tag T, you need to ensure that $P_e(W|T)$ produces a non-zero number no matter what word W you give it. You can use Laplace smoothing (as in MP 3) to fill in a probability for "UNKNOWN" which will be the return value for all words W that were not seen in the training data. For this initial implementation of Viterbi, use the same Laplace smoothing constant α for all tags.

This simple version of Viterbi will perform worse than the baseline code for provided dataset. However you should notice that it's doing better on the multiple-tag words. You should write this simple version of Viterbi under viterbi_p1 function in vertibi.py.

Part 2 Viterbi

The Part 1 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 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 and also beat the overall accuracy for both baseline and the simple vertibi model. You should write optimized version of Viterbi under viterbi_p2 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

- Tag 'X' rarely occurs in the dataset, large number of Laplace smoothing constant may overly smooth the emission probabilities and break your statistical computations. Small number of Laplace smoothing constant, e.g. 1e-5, may help.
- It's not advisable to use global variables in your implementation since the gradescope uses unittest functionality. For more information, refer to the brief introduction of unittest in course MP page.
- The autograder won't output useful information to help you debug. However, you may find helpful to compare your output sequence of tags with the ground truth sequence of tags in provided development set.

Extra Credit

The task for extra credit is to maximize the accuracy of the Viterbi code. You must train on only the provided brown training set (no external resources) and you should keep the basic Viterbi algorithm. However, you can make any algorithmic improvements you like.

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.

It is extremely hard to predict useful prefixes and suffixes from first principles. 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.

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 selected information from the first of the two tags. Also, beam search can be helpful to speed up decoding time.

The time out limitation for autograder is 10 minutes. The autograder is also enabled with leaderboard option. You can submit your code and compete with your classmates.

The grading criteria is the following: any submission with overall accuracy above or equal to 95.5% is considered as a valid submission.

- Students who ranked among **top 20%** of all valid submissions on Leaderboard will receive full **10** points.
- Students who ranked among **top 20%** to **top 50%** of all valid submissions on Leaderboard will receive **8** points.
- Students who made valid submissions will receive **5** points.

Code Submission

You should submit your **viterbi.py** file on gradescope for MP4 baseline, MP4 Viterbi part 1, and MP4 Viterbi part 2. You should submit yout **extra.py** file on gradescope for MP4 Extra Credit if you'd like to. If you have made local modifications to utils.py and/or mp4.py, make sure that your viterbi.py code works with the original provided versions. You are free to copy the same code from viterbi.py to extra.py if it's satisfied with extra credit grading criteria.