ECE448/CS440 Spring 2022

Assignment 4: HMM POS tagging

Due date: Wednesday March 28th, 11:59pm

Spring 2022: Haozhong Guan, Feiyu Zhang
Spring 2021: Kiran Ramnath, Mahir Morshed
Fall 2020: Amnon Attali, Jatin Arora
Spring 2020: Jialu Li, Guannan Guo, and Kiran Ramnath
Fall 2018: Margaret Fleck, Renxuan Wang, Tiantian Fang, Edward Huang
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 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.
- 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 traini data 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 report your accuracy.

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

You will need to write two main types of tagging functions:

- Baseline
- Viterbi: HMM tagger

The materials

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

- mp4.py (do not change)
- utils.py (do not change)
- baseline.py
- viterbi_1.py
- test_viterbi/
  - test_viterbi.py
  - utils.py (do not change)
- viterbi_2.py
- viterbi_ec.py

The code package also contains training and development data

- Brown corpus: data/brown-training.txt, data/brown-dev.txt

The provided code converts all words to lowercase. It also adds a START and END tag for each sentence when it loads the sentence. These tags are just for standardization. They will not be considered in accuracy computation.

You should use the provided 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 da from the same corpus to test your code after you submit it.

In addition, your code will be tested on two hidden datasets that are not available to you, which has different number of tags and words from the ones provided to you. **do NOT hardcode any of your important computations, such as initial probabilities, transition probabilities, emission probabilities, number or name of tags, and etc. We will inspect code for hardcoding computations/values and will penalize such implementations.**

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, viterbi_2, or viterbi_ec:

```
python3 mp4.py --train data/brown-training.txt --test data/brown-dev.txt --algorithm [baseline, viterbi_1, viterbi_2, viterbi_ec]
```

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.

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 two other datasets 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

Part 1: 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 ofte in training dataset.

A correctly working baseline tagger should get about 93.9% accuracy on the Brown corpus development set, with over 90% accuracy on multitag words and over 69% unseen words.

> **DO NOT ATTEMPT TO IMPROVE THE BASELINE ALGORITHM. IMPLEMENT IT EXACTLY AS DESCRIBED ABOVE.**

Part 2: Viterbi_1

IF YOU FIND THIS PART DIFFICULT TO BEGIN: You may find it easier to go to the next section and first implement only the Viterbi algorithm.

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 depen 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?)

You can assume that all sentences will begin with a START token, whose tag is START. So your initial probabilities will have a very restricted form, whether you choo to handcode appropriate numbers or learn them from the data. The initial probabilities shown in the textbook/texture examples will be handled by transition probabiliti from the START token to the first real word.

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 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 transition and emission 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 n matter what word W you give it. You can use Laplace smoothing (as in MP 2) 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 the Brown development dataset (somewhat over 93% accuracy). However you should not that it's doing better on the multiple-tag words (e.g. over 93.5%). You should write this simple version of Viterbi under viterbi_1 function in viterbi_1.py.

Starting or Debugging Part 2 Viterbi

THIS PART IS OPTIONAL AND SHOULD NOT BE SUBMITTED. However in case you find it difficult to directly attempt part 2, you may find it easier to start here. You can first implement or debug your Viterbi implementation alone using the dummy example provided. Jurafsky and Martin (section 8.4.6) provides a great worked out example to visualize the performance of the Viterbi algorithm. We provide that example as a sample test-case for you to ensure that your entire Viterbi algoirthm (Trellis computation and backtracking) is working correctly.

> **test_viterbi.py:** Write your Viterbi implementation here, and run it using `python3 test_viterbi.py`

Once you get this working (if you started Part 2 here), you can then extend it to the Brown dataset where you need to additionally compute for initial tag probabilities, unseen tags, and correct implementation of smoothing.

Part 3: 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, b 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 appear zero times in the training data (out-of-vocabulary or OOV words) and words that appear once in the training data (hapax words) tend to have simila parts of speech (POS). For this reason, instead of assuming that OOV words are uniformly distributed across all POS, we can get a much better estimate of their distribution by measuring the distribution of hapax words. Extract these words from the training data and calculate the probability of each tag on them. When you do y Laplace smoothing of the emission probabilities for tag T, scale the Laplace smoothing constant by P(T|hapax), i.e., the probability that tag T occurs given that the wo was hapax. Remember that Laplace smoothing acts by reducing probability mass for high-frequency words, and re-assigning some of that probability mass to low-frequency words. A large smoothing constant can end up skewing probability masses a lot, so experiment with small orders of magnitude for this hyperparameter.

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 th baseline on overall accuracy (e.g. 95.5%). You should write your optimized version of Viterbi under the viterbi_2 function in viterbi_2.py.

The hapax word tag probabilities may be different from one dataset to another, so your `viterbi_2` method should compute them dynamically from its training data ea time it runs.

Hints

- 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 statisti computations. A small value for the Laplace smoothing constant, e.g. 1e-5, may help.
- It's not advisable to use 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.

Extra Credit: Viterbi_ec

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

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 shoul 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 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 in Parts 2 and 3: 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" cou be mapped to "X-ING". Then you can use the hapax words to calculate suitable probability values for X-ING, as in Part 3.

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. You may assume that our completely hidden datasets are in English, so that word patterns from the Bro 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 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.

The autograder for parts 3 and ec has a timeout of 10 minutes.

Code Submission

As in previous MPs, the regular MP and the EC should be submitted separately. The regular MP includes the submission files **baseline.py**, **viterbi_1.py** and **viterbi_2.py**. The EC requires the submission file **viterbi_ec.py**.