CS440/ECE448 Fall 2021

Assignment 4: HMM POS tagging

Due date: Wednesday October 20th, 11:59pm

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 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 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 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 (in three versions)

The materials

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

- mp4.py (do not change)
- utils.py (do not change)
- baseline.py
- viterbi
  - viterbi_1.py
  - viterbi_2.py
  - viterbi_3.py

The code package also contains training and development data

- Brown corpus: data/brown-training.txt, data/brown-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, viterbi_2, or viterbi_3:

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

There are two places to submit on Gradescope, because testing can be a bit slow. The first submit point is for **baseline.py** and **viterbi_1.py**. The second is for **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.

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 93.9% accuracy on the Brown corpus development set, with over 90% accuracy on multitag words and over 69% on unseen words.

> **DO NOT ATTEMPT TO IMPROVE THE BASELINE CODE.**

Viterbi_1

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:

- 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 by backtracing.

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 a good choice for a smoothing method.

For example, to smooth the emission probabilities, consider each tag individually. For some 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," to use as the shared probability value for all words W that were not seen in the training data. The emission probabilities $P_e(W|T)$ should add up to 1 when we keep T fixed but sum over all words W (including UNKNOWN).

Now repeat the Laplace smoothing process to smooth the emission probabilities for all the other tags, one tag at a time. For this initial implementation of Viterbi, use the same Laplace smoothing constant α for all tags.

Similarly, to smooth the transition probabilities, consider one specific tag T. Use Laplace smoothing to fill any zeroes in the probabilities of which tags can follow T. The transition probabilities $P_e(T_b|T)$ should add up to 1 when we keep T constant and sum across all following tags T_b. Now repeat this process, replacing T with all the other possible first tags. The Laplace smoothing constant α should be the same for all first tags, but it might be different from the constant that you used for the emission probabilities.

This simple version of Viterbi will perform worse than the baseline code for the Brown development dataset (somewhat over 93% accuracy). However you should notice that it's doing better on the multiple-tag words (e.g. over 93.5%). You should write this simple version of Viterbi as the viterbi_1 function in viterbi_1.py.

DO NOT ATTEMPT TO IMPROVE VITERBI 1

Making Viterbi_1 Work Right

To get through all the parts of the MP, your Viterbi implementation must be computing the trellis values correctly and also producing the output tag sequence by backtracing. This is very much like the maze search in MP 1, where you searched forwards to get from the start to the goal, and then traced backwards to generate the best path to return. If you are producing the output POS sequence as you compute trellis values, you're using an incorrect ("greedy") method.

To help you debug your Viterbi implementation, we have provided a copy of the small example from Jurafsky and Martin (section 8.4.6). For this example, the initial, transition, and emission probabilities are all provided for you. You just need to copy in your trellis computation and your backtracking code.

THIS PART IS OPTIONAL. DO NOT SUBMIT IT ON GRADESCOPE.

The small example is in the template package, in the subdirectory test_viterbi.

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

There is also a small synthetic dataset in the data directory with files mttest-training.txt and mttest-dev.txt. Correctly working Viterbi code should get 100% on this.

When testing on the real dataset, you may find it helpful to make a very small file of test sentences. You can then print out the v and b values for each word. If you look at the emission probabilities or the v array values for a word, the correct tag should be one of the top couple values. Obviously unsuitable tags should have much lower values.

You can also print the matrix of tag transitions. It's not easy to be sure transitions should be highest. However, you can identify some transitions that should have very low values, e.g. transitions from anything to the START tag or to the X tag.

Some additional hints for making your code produce correct answers and run fast enough to complete on the autograder:

- Do probability computations using logs. Remember that math.log is faster than numpy.log.
- The trellis is described as an array in the lecture notes, but you can use any data structure to store it. Dictionaries work very well in Python.
- Avoid doing random access into python lists. Either use a "for" construction to walk down the list in order, or use some other data structure (e.g. a dictionary).
- In principle, the test data could contain a tag that you have not seen in the training data. Do not worry about this possibility because there is no way to guess the right answer. In a real linguistic situation, this would probably be due to a tagging error in the test data.
- If you suspect a bug in your backtracing code, it may be helpful to **temporarily** replace backtracing with the greedy method.
- Once you've computed a column of the Viterbi array, you do not need to keep the probability values from the previous columns. It's sufficient to keep the backpointer values.

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 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

- Tag X rarely occurs in the dataset.
- Setting a high value for 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.

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.