Part-of-Speech Tagging with Hidden Markov Models

Learning Goals

Once you complete this assignment, you should:

- Understand how to use hidden markov models and the Viterbi algorithm for part of speech tagging
- Develop an appreciation for the challenge of sparse data in probabalistic NLP models
- Gain additional experience with modification of hyperparameters to improve your model's performance
- Have experience performing error analysis for NLP

This assignment is connected to the following overall learning goals of the course:

- Be familiar with NLP methods in three key areas: text classification, text generation, and language understanding
- Explore various NLP applications, including applications with positive societal impact

Credits: this assignment is adapted from Rada Mihalcea's assignment on part of speech tagging.

Implementation Details

You should implement the Viterbi algorithm as described in the textbook. The starter code should help you out a lot - it includes code that will compute initialization log probabilities (`self._init_log_probs`), transition log probabilities (`self._transition_log_probs`), emission log probabilities (`self._emission_log_probs`), and the unique tags (`self._tags`) for you. In the pseudocode in the textbook, these are referred to as π, A, B, and Q, respectively (see slp section 8.4.2).

Your implementation should be similar to the psuedocode in the textbook, with a few modifications:

1. In the transition likelihoods, we implement laplace smoothing to avoid 0 probabilities. Instead of adding 1, add `self.k_transition` (see the starter code - this is called add-k smoothing, and we discussed it briefly when discussing n-gram language models). If `k = 0`, there is no smoothing. Specifically, $P_{smooth}(t_i|t_{i-1})$ can be computed as $\frac{C(t_{i-1},t_i)+k}{C(t_{i-1})+k*|Q|}$, where k is `self.k_transition` and $|Q|$ is the number of unique tags.
2. In your emission likelihoods, we add an unknown word using the tag `"<UNK>"` for each POS tag. The unknown word should have count `self.k_emission` (which may be a float). `self.k_emission` is also added to all other counts. Specifically, for all words where $C(t_i, w_i) > 0$, $P_{smooth}(w_i|t_i) = \frac{C(t_i,w_i)+k}{C(t_i)+k*(|W|+1)}$, where k is `self.k_emission` and $|W|$ is the number of words that appear with tag t_i. $P_{smooth}(\text{UNK}|t_i) = \frac{k}{C(t_i)+k*(|W|+1)}$.
   - When you see unknown word/tag pairs when executing the Viterbi algorithm, you should look up the probability of `"<UNK>"` associated with that tag.
3. Your calculations should all be performed in log space to avoid underflow. We'll convert probabilities of 0 to log probabilities of $-\infty$.

Many of these details are handled for you in the starter code.

Resources

Starter Code

The starter code for this assignment provides a main method and an abstract `POSTagger` class. Your main task is to fill in the `_predict_one` method of the `HMMPOSTagger` subclass. You might also edit the `__init__` method of the subclass. Do not edit any of the other classes.

The starter code also provides a baseline POS tagger, `BaselinePOSTagger`. The `train` method creates a dictionary that stores the most common tag for each token in the training data. That tag is chosen by the `_predict_one` method if the token appeared in the training data. If the token didn't occur in the training data, the most common tag overall is chosen.

Data Files

Two data files are available for you, `POS_train.txt` (training data) and `POS_dev.txt` (development data). The file has sentences separated by newlines. Each sentence is formatted as

```
tok_1/tag_1 tok_2/tag_2 ... tok_n/tag_n
```

The data is drawn from Universal Dependencies. It has been reformatted for our class to make it easier to process without external libraries.

Allowed External Resources

You may use any built-in python modules (in particular, you may think that Counter and defaultdict from the `collections` module are useful). You may not use external libraries.

While you may not search for code that solves this problem, you may search for ideas for improving your smoothing and handling of unknown words.

Deliverables

Code

You should submit a python script `model.py` on gradescope. Your main goal is to write a HMM part of speech tagger that **exceeds the accuracy of the baseline tagger**. This is actually not a trivial task, as the `BaselinePOSTagger` is a pretty good baseline.

Suggestions for Developing your Model To make sure that you have gotten the basic implementation of the Viterbi algorithm correct, I would recommend starting out by running `test_mini.py` which runs through two examples we have completed in class. You might need to debug by checking intermediate computations if your final results are incorrect.

Once those test cases are working, you can test your full model using `test.py` on the larger dataset from Universal Dependencies. If your model is working on the `test_mini.py` examples but doesn't exceed the baseline, that's expected - you need to test out different values for `k_emission` and `k_transition` to optimize your model!

Your tasks

The following components are required:

`_predict_one` method The `_predict_one` method should return a list of POS tags given a sentence (represented as a list of tokens). It should be implemented as described in slp section Figure 8.10. Here are a couple of things to be aware of in your implementation:

- While slp uses a matrix to represent the `viterbi` and `backpointer` arrays, using a dictionary or dictionaries is also an appropriate approach, and might be easier than using a list of lists.

- Your probabilities will be log probabilities, which means that you'll need to use + instead of * when the pseudocode shows multiplication of probabilities, because $log(M \cdot N) = log(M) + log(N)$. This is what we did in our examples in class!

`__init__` method There is already an `__init__` method provided in the code. However, once your write your code, you'll see that you don't necessarily improve upon the baseline with these arguments. To actually make an improvement, test out different values for the `k_transition` and `k_emission` arguments (try both higher and lower values than 1). Once you find something that works, **change the default values for `k_transition` and `k_emission` by updating these two lines of code**:

```
def __init__(self, k_transition: float = 1,
             k_emission: float = 1, extension: bool = False):
```

Possible Extensions (OPTIONAL)

If you'd like to learn more, you can try additional methods for handling unknown words (which would require updates to `train`). Here are some ideas: * Some types of words are more likely to be unseen in the training data than others - in particular, this includes open class words like nouns. One way to identify these types of words in a data-driven fashion is to count the number of words that appear only once with each POS tag - the more such words there are, the more likely it is for an unknown word to have that tag. Try to incorporate this observation in your model. These two webpages might be helpful: * Appendix: One-count smoothing in Jason Eisner's HMM assignment * Margaret Fleck's POS Tagging Assignment * Morphology might also give you some hints about the tag of an unknown word - for instance, words ending in `"ly"` are more likely to be adverbs. Try to incorporate this observation in your model. Here's a resource that might be helpful: * Morphological features help POS tagging of unknown words across language varieties * Finally, the TnT paper (one of our supplemental reading assignments) has some ideas for improving HMM POS taggers!

If you implement one of these methods, your code should still work as described in the assignment unless the argument `extension` is set to `True`in your `HMMPOSTagger`! Use conditionals to switch between the two functionalities. The `extension` flag will be used when testing your code for the leaderboard.

Report

In addition to your code, you must fill out the report in `report.md` (from the starter code).