Due Feb 26 by 11:59pm Points 25 Submitting a file upload Available Feb 15 at 12am - Feb 28 at 11:59pm 14 days

In this lab, you will experiment with HMM tagging. Your job is to implement, at a minimum, the Viterbi algorithm for a tagger, plus at least one extension. You may complete up to 10 extra bonus points by completing more extensions.

I've copied the Penn Treebank data into a more easy-to-use format for you, as well as provided some sample code for reading in the sentences. There is train, development, and test files (you can ignore development for this lab).

pos.zipPreview the document

Each line in each file contains one "sentence" (sometimes more than one sentence, but you can just treat it as one). Each line has a string of space-separated word/tag pairs, e.g.

The/DT file/NN contains/VB words/NNS ./.

Mandatory part (aka Part 0, 20 points): Implement a HMM bigram tagger using the Viterbi algorithm

- Read the word/tag pairs from pos_train.txt (example in create_counts.py) and count word given tag and tag given previous tag.
- Use these counts to compute probability distributions. Note that you will probably want to make probabilities in the log domain to prevent underflow.
- Implement the Viterbi algorithm to compute the most likely tag given the previous tag
- Using the sentences in pos_test.txt, compute the most likely tag for the words in each sentence. (Assume unknown words have some small probably of belonging to the most frequent tag.) Compare against the tags in the test set. Report the word-level tagging accuracy.

Extension 1 (15 points): Implement the forward-backward algorithm to estimate the probabilities of the tags assuming you did not have a labeled text -- that is:

- Initialization: Use the counts from pos_train.txt to figure out which tags are possible for words, and which bigram tags are possible, but then ignore the counts (i.e. make all counts equal to 1) to initialize the probabilities.
- Expectation: You should use the words from pos_train.txt but not the tags. Compute the probability of all tags for each word (given the sequence and alpha-beta recursion), as well as the bigram tag probability.
- Maximization: Use the tag probabilities as soft counts to update your P(word|tag) and P(tag|previous tag).
- Loop between the E and M steps until you converge (or pick a fixed number of iterations)
- Now use the Viterbi algorithm to compute the best tags for the test set. Compare the results to those you obtained from Part 0 (when you knew the tags). Is it better? Worse?
- HINT!!! If you test your code using Eisner's ice cream example, you can check your work against the spreadsheet. This will VASTLY simplify your debugging.

Extension 2 (5 points): Viterbi training - this is similar to forward-backward except you make a hard decision at every step. This should be much simpler.

- Initialization: Use the counts from pos_train.txt to figure out which tags are possible for words, and which bigram tags are possible, but then ignore the counts (i.e. make all counts equal to 1) to initialize the probabilities.
- Viterbi: You should use the words from pos_train.txt but not the tags. Use the Viterbi algorithm to find the most likely tag sequence for the training set words. Write out a file with the word/tag sequences similar to the pos_train.txt file (e.g. pos_train.iter1.txt).
- Estimation: Now read in the file you just wrote out using your count-based estimator to get new probability distributions.
- Loop between Viterbi and Estimation steps until you converge (or take a fixed number of steps)
- Now use the Viterbi algorithm to compute the best tags for the test set. Compare the results to those you obtained from Part 0 (when you knew the tags). Is it better? Worse?

Extension 3 (3 points): What happens if you make the words case-insensitive (both in training and test) using the model in Part 0?

Extension 4 (5 points): Extend Part 0 to implement a trigram tagger, including deleted interpolation to handle unseen trigram tags.

Extension 5 (5 points): Extend Part 0 to handle unseen words using a letter-based backoff.

Extension 6 (10 points): Implement your trained tagger in OpenFST. Hint: You can model P(W|T) as a FST, and P(T|T) as a FSA.

Submissions should include both code and a short document describing your findings.