Jake Traut

12/4/16

CSCI 3202 Assignment 8 HMM Report

Purpose:

The purpose of this assignment was to get familiar with implementing Hidden Markov Models (HMMs). Our goal was to build an HMM that successfully tags all part of speech of a given sentence. This involved calculating all possible transition and emission probabilities for the tags and words and applying these probabilities in the Viterbi algorithm.

Procedure:

In order to get the probabilities I created functions that took in lists filtered from the data in penntree.tag file as input, as well as making use of other global variables storing different useful information from the file. *LINES, TAGS,* and *WORDS* are all lists that store, as one could guess, the lines of words and tags, just the tags, and just the words, respectively. While processing the data from the file I also kept track of the frequency of the tags, stored in the dictionary *TAG_FREQ* with the tags being the key values, which was used for the probability calculations. These probabilities were stored in the globals *TRANS_PROB* and *EMISS_PROB* as dictionaries of dictionaries.

- **Transition Probabilities:**

  In order to get all transition probabilities between tags I created a function, *calc_trans(),* which takes a list of all possible tags, *allTags*, as input. I got the values for *allTags* by keeping a list, *TAGS*, of every occurrence of a tag (what comes right of the tab '\t') and setting *allTags* equal to the set of TAGS, which eliminates all duplicates in *TAGS* and leaves only the distinct tags. I also did this process to capture *WORDS* and *allWords.*

  Inside the *calc_trans()* function, I initialize a new 2D dictionary, *tag1tag2Freq*, and set all possible tag1, tag2 combinations to 0, such that I can incrementing the values by 1 when I begin counting, and also to help avoid any possible future invalid key entries. This was done with a double loop on *allTags* to enter in the 2D data. I then loop through TAGS and count all occurrences of the current tag following the previous tag. Then, in one more double loop of *allTags*, I populate the *TRANS_PROB[tag2][tag1]* 2D dictionary with the calculations of my *tag1tag2Freq* divided by the *TAG_FREQ[tag2].*

- **Emission Probabilities:**

My *calc_emiss()* function takes the two lists, *allWords* and *allTags*, as input. The function then initializes a new 2D dictionary, *wordTagFreq,* which will capture all the occurrences of a specific word being paired with a particular tag. For this all I needed was to loop through the global list LINES and increment the *wordTagFreq[lineWord][lineTag]* by 1 per valid line. The function then loops through a double loop of *allWords* and *allTags* in order to populate the *EMISS_PROB[tag][word]* dictionary with all the possible *wordTagFreq* divided by *TAG_FREQ* pairs.

- **Viterbi Algorithm:**

The implementation of the Viterbi algorithm provided in class actually worked just fine once I had all the probabilities populated and calculated properly. The algorithm accepts a list called *observations,* which is just the sentence broken down, and another *states* (equivalent to *allTags*), as well as all the necessary probabilities stored as *START_PROB, TRANS_PROB, EMISS_PROB* in 2D dictionaries.

The algorithm stores the probabilities and backtracking in an array of a dictionary appended with more dictionaries, creating a 3D matrix called *V*. It then searches for the best transition probability option and overall max probability for transition and emission. This is nice information but I didn't bother printing any of it, as well as cutting out some of the extra printing statements from the original implementation. The translations from original sentence (or observation) to part of speech tags occurs next and is stored in a list called *opt.* It first finds the most probable ending state and begin to backtrack from there following the next most probable until reaching the beginning of the observation.

Data:

The baseline data for this assignment was stored in a file called penntree.tag that is filled with thousands of sentences separated by empty lines. The format is one Word Tag pair per line, being separated by a tab '\t'. In order to pre-process the data, I opened the file as sentences and began organizing the data line by line into three separate lists, *LINES, TAGS* and *WORDS* as well as keeping track of tag frequencies with the dictionary *TAG_FREQ[tag].* With knowledge of the data's format, Word '\t' Tag, I just partitioned the line strings on the '\t' value and captured what comes after the tab as a tag and what came before the tab as the word, then appended these values into their respective lists, as well as counted the tag frequencies here. Converting the *TAGS* and *WORDS* lists into sets and setting them equal to *allTags* and *allWords* respectively also made the data much more useful in having the lists of all unique tags and words without repeats.

Results:

Using the much needed "sanity checks" throughout the write-up as reference of accuracy and correctness, I could confirm I was getting the right numbers in my probabilities and outputs in my part of speech tagger on the two test sentences. No, I take that back, the code is exceptional and works completely fine on the other more complicated example sentences too. I was very excited to see this on the first run through these sentences. I can confirm the correctness of these tags as they are shown on the write-up, but I don't think my personal observations can be totally valid in staying they are correct when writing my own sentences, as I don't know what the actual correct tags would be either. I'd like to think I can trust my work though, and it's probably more correct in producing tags than I could guess on my own. Hidden Markov Models are pretty neat.