NLP 201: Assignment 3

Ethan Sin

December 15, 2023

1 Deriving Viterbi

1.1 Question

Since maximizing $P(t|w)$ and $P(t,w)$ are equivalent in nature, you can also apply $log()$ to it as the values that result in the maximizing will remain the same too. Therefore, since all changes being applied here will result in the same t being chosen by maximizing, changing $argmax_t P(t|w)$ to $argmax_t log(P(t,w))$ will also still result in the same t for maximizing.

1.2 Question

$$\pi_j(t_j) = \max_{t_1 \dots t_{j-1}} \sum_{i=1}^{j} score(w, i, t_i, t_{i-1})$$

$$\pi_{j-1}(t_{j-1}) = \max_{t_1 \dots t_{j-2}} \sum_{i=1}^{j} score(w, i, t_i, t_{i-1})$$

Substitute the second line into the first line as the equivalent to the first with one lower j.

$$\pi_{j-1}(t_{j-1}) = \max_{t_{j-1}} [\max_{t_1 \dots t_{j-2}} [score(w, j, t_j, t_{j-1}) + \sum_{i=1}^{j} score(w, i, t_i, t_{i-1})]]$$

$$\max_{t_{j-1}} [score(w, j, t_j, t_{j-1}) + \max_{t_1 \dots t_{j-2}} [\sum_{i=1}^{j} score(w, i, t_i, t_{i-1})]]$$

$$\max_{t_1 \dots t_{j-2}} [\sum_{i=1}^{j} score(w, i, t_i, t_{i-1})]$$

$$\downarrow$$

$$\pi_{j-1}(t_{j-1})$$

$$\max_{t_{j-1}} score(w, j, t_j, t_{j-1}) + \pi_{j-1}(t_{j-1})$$

1.3 Question

A brute force method can be used to check every possible path and sum the scores over every possible tag sequence given the sequence of words. The time complexity would be $O(wt^2)$ as for every token through the sequence it would have to calculate a score for the number of transitions coming into the current state times the number of transitions going out of the current state, both of which are equal to the number of possible tags.

1.4 Question

$$\pi_j(t_j) = \bigoplus_{t_1 \ldots t_{j-1}} \bigotimes_{i=1}^{j} score(w, i, t_i, t_{i-1})$$

$$\pi_{j-1}(t_{j-1}) = \bigoplus_{t_1 \ldots t_{j-2}} \bigotimes_{i=1}^{j} score(w, i, t_i, t_{i-1})$$

Substitute the second line into the first line as the equivalent to the first with one lower j.

$$\pi_{j-1}(t_{j-1}) = \bigoplus_{t_{j-1}} [\bigoplus_{t_1 \ldots t_{j-2}} [score(w, j, t_j, t_{j-1}) + \bigotimes_{i=1}^{j} score(w, i, t_i, t_{i-1})]]$$

$$\bigoplus_{t_{j-1}} [score(w, j, t_j, t_{j-1}) + \bigoplus_{t_1 \ldots t_{j-2}} [\bigotimes_{i=1}^{j} score(w, i, t_i, t_{i-1})]]$$

$$\bigoplus_{t_1 \ldots t_{j-2}} [\bigotimes_{i=1}^{j} score(w, i, t_i, t_{i-1})]$$

$$\downarrow$$

$$\pi_{j-1}(t_{j-1})$$

$$\bigoplus_{t_{j-1}} score(w, j, t_j, t_{j-1}) + \pi_{j-1}(t_{j-1})$$

2 Hidden Markov Model

The training step of a Hidden Markov Model with Viterbi decoding involves creating tables of transition and emissions. In the case of this dataset, the transitions are probabilities of bigram sequences between part-of-speech tags, and the emissions are probabilities of a specific token given the current part-of-speech tag. The end goal will be to use a Viterbi algorithm along with these tables to decode an input sequence of tokens to output a sequence of part-of-speech tags for those tokens.

My transition and emission tables exist as NumPy arrays, shape (number of unique tags in the train set, number of unique tags in the train set) and (number of unique tags in the train set, number of unique tokens in the train set). The tables are constructed first by iterating over the train set and counting in two dictionary how many times a tag appears compared to how many times that tag is tied to a specific token, and how many times a

tag appears compared to how many times another tag follows it. For every unique tag and token, there are also two dictionaries to retrieve a tag or token by index and vice versa. This will be useful to retrieve the correct transition and emission probabilities while creating the Viterbi matrices later on. A loop is then constructed to update every value in the NumPy arrays using a probability calculation function that also implements adjustable add-α smoothing. Probabilities for a tag transiting from a <START> tag or to a <STOP> tag are also considered. In the data, there also existed 29 instances of unique tags that looked like "NN|JJ". In these cases the tag was treated as only the first tag, in this case: "NN".

3 Viterbi Decoding

Now that the transition and emission tables are created, the Viterbi algorithm will be implemented and used to decode the input from our train and test sets. For my code, I implemented the algorithm from Jurafsky(Jurafsky and Martin, 2019) as closely as possible. This aims to be a more efficient way of finding the most likely path through a Hidden Markov Model using backpointers and eliminating low-likelihood routes as early as possible.

First, I initialized the Viterbi matrix of shape (number of observations, number of states) as a matrix of zeroes using NumPy and an identical one for storing backpointers. Then, I initialized the first set of states tied to the first observation as simply the probabilities of start transiting to the appropriate state, each multiplied by the emission probability of that state emitting the first observation. The backpointers would all remain 0 here as there is no possibility for transiting from anything other than <START>. The rest of the decoding loop is a nested loop going through each observation and each state within it where each time an observation is fully looped through is a timestep. The loop will then update the current value at the indices given by the loop by finding the maximum value of the Viterbi calculated at the states of the previous timesteps, and assign a backpointer to the index of the state it found that maximum value for. It will then conclude by finding the maximum value and index at the last timestep, and create a path following the backpointers to the start. This can then be easily decoded by accessing by indices which tags the values in the array belong to, which is then used to pair with the observation token in the output.

4 Evaluation

The model was tested on the dev set with different α values. As expected, a lower α resulted in better results. This was expected because as the alpha gets lower, it allows the distribution to better represent the original data. A higher value begins to 'flatten' the distribution, making originally low probabilities much higher. Table 1 below displays the results of the testing I did.

After doing the testing on the dev set, I compared the performance of my tagger to the baseline model constructed by simply assigning the most likely tag to a given word, and to NLTK's POS tagger. The results are shown in Table 2.

The HMM tagger is surprisingly not much better than the baseline model in my case, but

α	F1	Precision	Recall
1	.60	.67	.58
0.5	.62	.67	.60
0.1	.64	.66	.63
1.5	.59	.65	.57
2	.57	.65	.54

Table 1: Macro Averages at different α values on the dev set

Model	F1	Precision	Recall
Baseline	.71	.74	.70
NLTK	.77	.79	.77
0.1 α	.74	.76	.73

Table 2: Macro Averages of 0.1 α compared to NLTK and baseline on the test set

the surprise here is more attending to the efficacy of the baseline. With some thought, this is somewhat expected however. Most words are likely to be their most likely tag according to Zipf's law, so the accuracy from the baseline model should be mostly accurate. The HMM tagger is doing its job by covering more of the edge cases where a word's part of speech is more likely something else according to its placement in the sentence. Perhaps to improve the model the transitions can expand beyond bigrams, if in some cases a word is more likely to be a certain part of speech given a longer context than bigram. The decoding step on my machine already runs at 30 seconds to tag the dev set and test sets, so I am not sure what else can be done to improve the processing time.

References

Jurafsky, D. and Martin, J. H. (2019). *Speech and Language Processing.* Pearson.