TTIC 31210: Advanced Natural Language Processing
Assignment 3: Structured Prediction (60 points)

Instructor: Kevin Gimpel
Assigned: Wednesday, May 1, 2019
Due: 7:00 pm, Thursday, May 16, 2019
Submission: email to `kgimpel@ttic.edu`

Submission Instructions

Package your report and code in a single zip file or tarball, name the file with your first and last name followed by "_hw3", and email the file to `kgimpel@ttic.edu` by the due date and time. For the report, use pdf format or a Jupyter Notebook. Please do not use plain text files.

Collaboration Policy

You are welcome to discuss assignments with others in the course, but solutions and code should be written individually. You may use software libraries for helper code, but you must write the code for the inference algorithms yourself.

Overview

In this assignment, you will do part-of-speech tagging using some of the methods you've learned from the structured prediction segment of the course. You should submit a report with your results as well as your code.

Data

You will be using manually-annotated English data from the Universal Dependencies project. I did some preprocessing of the data for you. The data available from the course web page contains the following two files:

- `en_ewt.train`: training data; contains 12,543 annotated sentences.
- `en_ewt.dev`: development data; contains 2,002 annotated sentences.

Each non-blank line corresponds to a single token in a sentence and follows the format "word[TAB]tag". Blank lines separate sentences. I did not give you the test set. Since there is not much to tune in this assignment, you will just be reporting all of your results on the development set (DEV).

I replaced approximately 3% of singleton words (i.e., words that appeared once in the training data) with the word "UNKNOWN". In DEV, I replaced all words that were not in the training vocabulary with "UNKNOWN". For this assignment, just treat "UNKNOWN" like any other word.

Hidden Markov Models for Tagging

We will use a hidden Markov model (HMM) for part-of-speech tagging. We will use Y_t to denote the random variable corresponding to the label (tag) at position t. We will use X_t to denote the random variable corresponding to the symbol (word) at position t. A typical HMM uses the following conditional independence assumptions:

$$\text{for } k > 1 : Y_t \perp\!\!\!\perp Y_{t-k} \mid Y_{t-1}$$
$$\text{for } k > 1 : Y_t \perp\!\!\!\perp Y_{t+k} \mid Y_{t+1}$$
$$\text{for } k \neq 0 : X_t \perp\!\!\!\perp Y_{t+k} \mid Y_t$$

We will use $\boldsymbol{Y}$ to denote the full sequence of Y random variables where individual random variables are indexed as Y_t. We will use lowercase letters to denote values of random variables, i.e., y is a value that Y_t can take on and $\boldsymbol{y}$ is a value that $\boldsymbol{Y}$ can take on. We use analogous notation for the X random variables. The set of possible labels (tags) will be denoted $\mathcal{L}$ and the set of possible symbols (words) will be denoted $\mathcal{V}$.

Using these conditional independence assumptions, the probability of a length-T sequence of labels and symbols is:

$$P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y}) = P(\texttt{</s>} \mid Y_T = y_T) \prod_{t=1}^{T} P(Y_t = y_t \mid Y_{t-1} = y_{t-1}) P(X_t = x_t \mid Y_t = y_t)$$

where $\texttt{</s>}$ is the end-of-sequence label and where we fix $Y_0 = \texttt{<s>}$, the start-of-sequence label.

We parameterize the probability distributions in the above equation as follows. The parameters of the $P(Y_t \mid Y_{t-1})$ distributions are called the **transition probabilities** and we denote the probability of transitioning from tag y' to tag y as $p_{\boldsymbol{\tau}}(y \mid y')$. The parameters of the $P(X_t \mid Y_t)$ distributions are called the **emission probabilities** and we denote the probability of emitting word x from tag y as $p_{\boldsymbol{\eta}}(x \mid y)$. So, we write

$$P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y}) = p_{\boldsymbol{\tau}}(\texttt{</s>} \mid y_T) \prod_{t=1}^{T} p_{\boldsymbol{\tau}}(y_t \mid y_{t-1}) p_{\boldsymbol{\eta}}(x_t \mid y_t) \tag{1}$$

1. Supervised Parameter Estimation for HMMs (15 points)

- **(a) (10 points)** Implement supervised learning for an HMM for POS tagging. Use maximum likelihood estimation for learning, i.e., "count and normalize", along with add-λ smoothing:

$$p_{\boldsymbol{\tau}}(y \mid y') \leftarrow \frac{\text{count}(\langle y', y\rangle) + \lambda_\tau}{\sum_{y''} (\text{count}(\langle y', y''\rangle) + \lambda_\tau)} \tag{2}$$

$$p_{\boldsymbol{\eta}}(x \mid y) \leftarrow \frac{\text{count}(\langle y, x\rangle) + \lambda_\eta}{\sum_{x'} (\text{count}(\langle y, x'\rangle) + \lambda_\eta)} \tag{3}$$

  where $\text{count}(\langle y', y\rangle)$ is the number of times in the training set that the tag bigram "y' y" was observed, and $\text{count}(\langle y, x\rangle)$ is the number of times in the training set that word x was tagged with tag y. The above equations use "add-λ" smoothing with different values of λ for each distribution. Use $\lambda_\tau = 0.1$ and $\lambda_\eta = 0.001$ in your implementation.

There are some impossible events that should always have probability zero and never receive any amount of probability mass from smoothing. They are as follows:

$$p_{\boldsymbol{\tau}}(\texttt{</s>} \mid \texttt{<s>}) = 0 \text{ (i.e., empty sequences are not permitted!)}$$
$$p_{\boldsymbol{\tau}}(\texttt{<s>} \mid y) = 0, \text{ for all } y$$
$$p_{\boldsymbol{\tau}}(y \mid \texttt{</s>}) = 0, \text{ for all } y$$
$$p_{\boldsymbol{\eta}}(x \mid \texttt{<s>}) = 0, \text{ for all } x$$
$$p_{\boldsymbol{\eta}}(x \mid \texttt{</s>}) = 0, \text{ for all } x$$

Note: There should be no unknown words in the development data because all of them were replaced by "UNKNOWN". Nonetheless, smoothing is still very helpful, because not all tag transitions are seen in the training data and not all tag-word emissions are seen in the training data.

There are 50 POS tags. Each of the 50 can transition to each of the other 50 as well as </s>, then there are 50 tags that <s> can transition to, giving us (50 $\times$ 51) + 50 = 2,600 transition parameters.

The vocabulary size, including UNKNOWN, is 19,380. So, there are 19,380 $\times$ 50 = 969,000 emission parameters.

These statistics and part (b) below are provided to help you ensure you get the implementation correct for this part. This is important because all of the parts below depend on it, as does Assignment 4.

Note: to avoid underflow in the inference algorithms below, you will probably need to implement estimation and inference in the log domain. That is, instead of storing probabilities, store log-probabilities. When you would ordinarily multiply together two probabilities, instead just add their log-probabilities. When you would ordinarily add together two probabilities, instead log-add their log-probabilities. The log-add function can be found in standard libraries or pseudocode can be found online.

Submit your code.

- **(b) (5 points)** After estimating the parameters using add-λ smoothing as described above, print the 5 tags with the highest probability of following <s>, i.e., the y's with the highest values of $p_{\boldsymbol{\tau}}(y \mid \texttt{<s>})$, along with their probabilities. (Hint: with my implementation, the most probable is PRP with probability 0.224506 and second is NNP with probability 0.115963.)

  Print the top 10 most probable words emitted by the adjective tag ("JJ"), along with their probabilities. (Hint: using my implementation, the most probable word is "other" which has probability 0.023069.)

2. Preliminaries for Inference with HMMs (5 Points)

For supervised HMMs, there is no inference required during learning. However, to use the model to tag a new sentence $\boldsymbol{x}$, we need to solve the following argmax inference problem:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\operatorname{argmax}}\ P(\boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{X} = \boldsymbol{x}) = \underset{\boldsymbol{y}}{\operatorname{argmax}}\ P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y}) \quad (4)$$

Typically, the Viterbi algorithm is used to exactly solve this inference problem. Below you will implement and experiment with Viterbi. But first you will experiment with simpler methods.

- **(a) Log-probability calculation (2 points):** Implement a function that computes the log-probability of the development sentences and corresponding tag sequences, i.e., the natural logarithm of Eq. 1, summed over labeled sentences $\{\langle \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\rangle\}$ in the development set:

$$\sum_i \log P(\boldsymbol{X} = \boldsymbol{x}^{(i)}, \boldsymbol{Y} = \boldsymbol{y}^{(i)})$$

  Report the log-probability of the sentences in DEV with their gold standard tag sequences. (Hint: using my implementation, I found a log-probability of -169936.)

- **(b) Local predictor baseline (3 points):** A very simple (and very bad) inference algorithm would simply ignore the transition distributions and choose label t as follows:

$$\hat{y}_t = \operatorname*{argmax}_{y \in \mathcal{L}} \; p_{\boldsymbol{\eta}}(x_t \mid y)$$

  Implement this decoder. Report the tagging accuracy on DEV, the total time required for this procedure to produce predictions on DEV, and the log-probability of the sentences in DEV with the predicted tag sequences from this local algorithm.

3. Greedy Left-to-Right Algorithms (10 points)

A better way to do inference would be to use a greedy left-to-right procedure. The algorithm is described mathematically below, where we use $G(t)$ to denote the predicted tag at position t:

$$\begin{aligned} & G(0) = \texttt{<s>} \\ 0 < t < T: \; & G(t) = \operatorname*{argmax}_{y \in \mathcal{L}} \; p_{\boldsymbol{\eta}}(x_t \mid y) p_{\boldsymbol{\tau}}(y \mid G(t-1)) \\ & G(T) = \operatorname*{argmax}_{y \in \mathcal{L}} \; p_{\boldsymbol{\eta}}(x_T \mid y) p_{\boldsymbol{\tau}}(y \mid G(T-1)) p_{\boldsymbol{\tau}}(\texttt{</s>} \mid y) \end{aligned}$$

This algorithm produces predicted tags in entries $G(1), ..., G(T)$.

- **(a) (10 points):** Implement this algorithm and submit your code. Report the tagging accuracy on DEV, the total time required for this procedure to produce predictions on DEV, and the log-probability of the sentences in DEV with the predicted tag sequences from this algorithm. This algorithm should have a higher accuracy than the local algorithm, and produce outputs with higher log-probability, though it will be slower.

4. Greedy Right-to-Left Algorithms (10 points)

Above we chose a greedy left-to-right inference strategy. How about using the same idea of greedy inference, but in a right-to-left order? The algorithm is described below, where we use $R(t)$ to denote the predicted tag at position t:

$$\begin{aligned} & R(T+1) = \texttt{</s>} \\ 1 < t \leq T: \; & R(t) = \operatorname*{argmax}_{y \in \mathcal{L}} \; p_{\boldsymbol{\eta}}(x_t \mid y) p_{\boldsymbol{\tau}}(R(t+1) \mid y) \\ & R(1) = \operatorname*{argmax}_{y \in \mathcal{L}} \; p_{\boldsymbol{\eta}}(x_1 \mid y) p_{\boldsymbol{\tau}}(R(2) \mid y) p_{\boldsymbol{\tau}}(y \mid \texttt{<s>}) \end{aligned}$$

This algorithm produces predicted tags in entries $R(1), ..., R(T)$.

- **(a) (5 points):** Implement this algorithm and submit your code. Report the tagging accuracy on DEV, the total time required for this procedure to produce predictions on DEV, and the log-probability of the sentences in DEV with the predicted tag sequences.
- **(b) (5 points):** Discuss the results. In your discussion, consider the following points. What do you notice about the performance (accuracy and log-probability) of this algorithm compared to the local and left-to-right algorithms? Why do you think you see these trends? What changes might you make to improve the accuracy of greedy right-to-left algorithms?

5. Exact Inference with Viterbi (20 points)

The Viterbi recursive equations for bigram HMMs are defined:

$$\begin{aligned} V(1,y) &= p_{\boldsymbol{\eta}}(x_1 \mid y)\, p_{\boldsymbol{\tau}}(y \mid \texttt{<s>}) \\ 1 < t \le T: \quad V(t,y) &= \max_{y' \in \mathcal{L}}\; p_{\boldsymbol{\eta}}(x_t \mid y)\, p_{\boldsymbol{\tau}}(y \mid y')\, V(t-1,y') \end{aligned}$$

Given a sentence $\boldsymbol{x}$ with length $|\boldsymbol{x}| = T$, the final value $goal(\boldsymbol{x})$ is:

$$goal(\boldsymbol{x}) = \max_{y' \in \mathcal{L}}\; p_{\boldsymbol{\tau}}(\texttt{</s>} \mid y')\, V(T,y')$$

When using memoization, the space complexity is $O(T|\mathcal{L}|)$ and the time complexity is $O(T|\mathcal{L}|^2)$.

Intuitively, think of $V(t,y)$ as answering the question: what is the score of the best way to get to position t ending in label y? The answer to that question involves looping over all possible labels y' at position $t-1$ and, for each y', considering the best way to get to position $t-1$ ending in label y', which is represented by $V(t-1,y')$.

The above algorithm computes the probability of the label sequence with the largest probability, but does not return the label sequence itself. We can augment the above algorithm with another data structure to record the y's that yielded the maximum probabilities in each step:

$$\begin{aligned} 1 < t \le T: \quad L(t,y) &= \operatorname*{argmax}_{y' \in \mathcal{L}}\; p_{\boldsymbol{\eta}}(x_t \mid y)\, p_{\boldsymbol{\tau}}(y \mid y')\, V(t-1,y') \\ \hat{y}_T &= \operatorname*{argmax}_{y' \in \mathcal{L}}\; p_{\boldsymbol{\tau}}(\texttt{</s>} \mid y')\, V(T,y') \end{aligned}$$

Note that we have essentially just replaced $\max$ with argmax. Think of $L(t,y)$ as containing the best previous label for getting to position t ending in label y. The *reversed* sequence of predicted labels is then $\langle \hat{y}_T, L(T,\hat{y}_T), L(T-1, L(T,\hat{y}_T)), L(T-2, L(T-1, L(T,\hat{y}_T))), ... \rangle$. We can also write this as follows:

$$\begin{aligned} &\hat{y}_T \\ &\hat{y}_{T-1} = L(T,\hat{y}_T) \\ &\hat{y}_{T-2} = L(T-1,\hat{y}_{T-1}) \\ &... \\ &\hat{y}_2 = L(3,\hat{y}_3) \\ &\hat{y}_1 = L(2,\hat{y}_2) \end{aligned}$$

where $\hat{y}_t$ is the predicted label at position t. Note that $L(t,y)$ is not defined for $t = 1$.

Implement the Viterbi algorithm and submit your code. You can use a "bottom-up" or a "top-down" implementation (they produce equivalent results). In both cases, you will need a table of computed values $V(t, y)$ indexed by both the position t and label y. For the top-down version, implement the recursive equations directly as defined above; however, before computing a particular $V(t, y)$, you should check to see if you have already computed it and, if so, simply return its value. Then simply compute $goal(\boldsymbol{x})$ and all values of $V(t, y)$ will be computed and filled in. By contrast, the bottom-up version uses a left-to-right ordering of the computations for filling in the values of $V(t, y)$ and ultimately for calculating $goal(\boldsymbol{x})$. That is, the bottom-up algorithm first computes and stores $V(1, y)$ for all y, then computes and stores $V(2, y)$ for all y (this step uses $V(1, y)$), then computes and stores $V(3, y)$ for all y (this step uses $V(2, y)$, which in turn uses $V(1, y)$), etc.

Run Viterbi on each sentence in DEV. As above, report the tagging accuracy, the total time required to run Viterbi on the sentences in DEV, and the log-probability of the sentences in DEV with the predicted tag sequences from Viterbi. (You should find that the gold tag sequences have lower log-probability than the outputs from the greedy left-to-right and Viterbi algorithms. This shows that there is **model error**. You should also find that the greedy outputs have lower log-probability than the Viterbi output; the gaps reveal the amount of **search error** of the greedy algorithms.)

6. Extra Credit: Beam Search for HMMs (5 points)

Let's now define beam search for a bigram HMM with beam width b. We first define an **item** as a tuple $\langle y, v, i\rangle$ where $y \in \mathcal{L}$, $v \in \mathbb{R}$, and $i \in [b]$, where the notation $[b]$ denotes the set containing the first b positive integers. The label y is the most recent label in the item, the value v is the score of the item, and the index i is a "backpointer" to an item from the preceding timestep. The beam search algorithm will build and manipulate sets and lists of items. The base case follows:

$$A(1) = \{\langle y, p_{\boldsymbol{\eta}}(x_1 \mid y)\, p_{\boldsymbol{\tau}}(y \mid \text{<s>}), -1\rangle : y \in \mathcal{L}\}$$
$$B(1) = \text{max-}b(A(1))$$

where $A(1)$ is a set of items and $B(1)$ is an indexable list of items. We choose "A" to connote "All" and "B" to connote "Beam". First, $A(1)$ is generated by producing the items corresponding to using label y at position 1 in the sequence for all possible labels $y \in \mathcal{L}$, designating -1 as the (null) backpointer. The max-b function takes a set of items as input and returns a list containing the b items with the largest values. Note that the items in the list $B(1)$ do *not* have to be sorted according to their values. The ordering of the items in the list is arbitrary. The only requirements are that $B(1)$ contains the b items from $A(1)$ with the highest values, and the items in $B(1)$ are accessible with integer indices $i \in [b]$. We require items to be accessible with integer indices so that we can store and follow backpointers. The case for $1 < t < T$ is as follows:

$$1 < t < T: \quad A(t) = \{\langle y,\ p_{\boldsymbol{\eta}}(x_t \mid y)\, p_{\boldsymbol{\tau}}(y \mid y')\, v',\ j\rangle : y \in \mathcal{L}, \langle y', v', i'\rangle \text{ is element } j \text{ in } B(t-1)\}$$
$$1 < t < T: \quad B(t) = \text{max-}b(A(t))$$

That is, the set $A(t)$ is created by considering all combinations of labels $y \in \mathcal{L}$ and previous items (elements of $B(t-1)$), where the score of a new item includes a multiplication by the score v' of the previous item. The backpointer in the new item is the index of the previous item in the previous list. The backpointer of the previous item, i', is not used in creating the new item.

Then $B(t)$ is created by keeping the top-scoring b hypotheses from $A(t)$. Again, the items in $B(t)$ must be accessible using indices but they do not need to be sorted.

The final case is below:

$$A(T) = \{\langle y,\ p_{\boldsymbol{\eta}}(x_T \mid y)\, p_{\boldsymbol{\tau}}(y \mid y')\, p_{\boldsymbol{\tau}}(\text{</s>} \mid y)\, v',\ j\rangle : y \in \mathcal{L}, \langle y', v', i'\rangle \text{ is element } j \text{ in } B(T-1)\}$$
$$B(T) = \text{max-1}(A(T))$$

There are two differences in the final case. First, we use the end-of-sequence transition probability in the score of the new items created. Second, we use $\max\text{-}1$ instead of $\max\text{-}b$ because we only care about the highest-scoring sequence. If we wanted to output the b highest-scoring sequences, we should use $\max\text{-}b$ in this step (and probably sort them by score).

The space and time complexity of this algorithm will depend on the algorithms used for $\max\text{-}b$ and $\max\text{-}1$. Therefore, for large values of b, beam search may be slower than Viterbi. When $b = 1$, beam search reduces to the greedy algorithm described above.

The above algorithm yields $B(T)$ with a single item $\langle y, v, i\rangle$, where y is the final label in the label sequence found, v is the score of the highest-scoring label sequence found, and i is a backpointer used to reconstruct the full label sequence as follows:

$$\begin{aligned}
&\hat{y}_T = y, i_T = i, \text{ where } \langle y, v, i\rangle \text{ is element 1 of } B(T)\\
&\hat{y}_{T-1} = y, i_{T-1} = i, \text{ where } \langle y, v, i\rangle \text{ is element } i_T \text{ of } B(T-1)\\
&\hat{y}_{T-2} = y, i_{T-2} = i, \text{ where } \langle y, v, i\rangle \text{ is element } i_{T-1} \text{ of } B(T-2)\\
&\dots\\
&\hat{y}_2 = y, i_2 = i, \text{ where } \langle y, v, i\rangle \text{ is element } i_3 \text{ of } B(2)\\
&\hat{y}_1 = y, \text{ where } \langle y, v, i\rangle \text{ is element } i_2 \text{ of } B(1)
\end{aligned}$$

where $\hat{y}_t$ is the predicted label at position t.

Implement this beam search algorithm and run it on DEV for beam widths $b \in \{1, 2, 3, 4, 5, 10, 20, 50, 1000\}$. For each b value, report the accuracy, time, and log-probability, as you did for the previous algorithms. When using $b = 1$, the log-probability and accuracy should match those of the greedy left-to-right algorithm.