Danilo Dordevic: Assignment 2

ddordevic@student.ethz.ch, 21-954-888.

14/12/2022 - 17:21h

Question 1

a) 1. $< \mathbb{R} \times \mathbb{R}, \oplus, \mathbf{0} >$ is a commutative monoid:

- Identity element:

$$\begin{aligned} \boldsymbol{a} \in \mathbb{R} \times \mathbb{R}, \boldsymbol{a} &= (a_1, a_2), a_1 \in \mathbb{R}, a_2 \in \mathbb{R} \\ \boldsymbol{a} \oplus \mathbf{0} &= (a_1, a_2) \oplus (0, 0) \\ &= (a_1 + 0, a_2 + 0) \\ &= (0 + a_1, 0 + a_2) \\ &= \mathbf{0} + \boldsymbol{a} \\ &= (a_1, a_2) = \boldsymbol{a} \end{aligned}$$

- Associativity of $\oplus$:

$$\begin{aligned} (\boldsymbol{a} \oplus \boldsymbol{b}) \oplus \boldsymbol{c} &= [(a_1, a_2) \oplus (b_1, b_2)] \oplus (c_1, c_2) \\ &= (a_1 + b_1, a_2 + b_2) \oplus (c_1, c_2) \\ &= (a_1 + b_1 + c_1, a_2 + b_2 + c_2) \\ &= (a_1 + (b_1 + c_1), a_2 + (b_2 + c_2)) \\ &= (a_1, a_2) \oplus (b_1 + c_1, b_2 + c_2) \\ &= (a_1, a_2) \oplus [(b_1, b_2) \oplus (c_1, c_2)] \\ &= \boldsymbol{a} \oplus (\boldsymbol{b} \oplus \boldsymbol{c}) \end{aligned}$$

- Commutativity of $\oplus$:

$$\begin{aligned} \boldsymbol{a} \oplus \boldsymbol{b} &= (a_1, a_2) \oplus (b_1, b_2) \\ &= (a_1 + b_1, a_2 + b_2) \\ &= (b_1 + a_1, b_2 + a_2) \\ &= (b_1, b_2) \oplus (a_1, a_2) \\ &= \boldsymbol{b} \oplus \boldsymbol{a} \end{aligned}$$

2. $< \mathbb{R} \times \mathbb{R}, \otimes, \mathbf{1} >$ is a monoid:

- Identity element:

$$\begin{aligned}\boldsymbol{a} \otimes \mathbf{1} &= (a_1, a_2) \otimes (1, 0) \\ &= (a_1 \cdot 1, a_1 \cdot 0 + a_2 \cdot 1) \\ &= (a_1, a_2) \\ &= \boldsymbol{a}\end{aligned} \qquad \begin{aligned}\mathbf{1} \otimes \boldsymbol{a} &= (1, 0) \otimes (a_1, a_2) \\ &= (1 \cdot a_1, 1 \cdot a_2 + 0 \cdot a_1) \\ &= (a_1, a_2) \\ &= \boldsymbol{a}\end{aligned}$$

- Associativity of $\otimes$:

$$\begin{aligned}(\boldsymbol{a} \otimes \boldsymbol{b}) \otimes \boldsymbol{c} &= [(a_1, a_2) \otimes (b_1, b_2)] \otimes (c_1, c_2) \\ &= (a_1 b_1, a_1 b_2 + a_2 b_1) \otimes (c_1, c_2) \\ &= (a_1 b_1 c_1, a_1 b_1 c_2 + (a_1 b_2 + a_2 b_1) \cdot c_1) \\ &= (a_1 b_1 c_1, a_1 b_1 c_2 + a_1 b_2 c_1 + a_2 b_1 c_1) \\ &= (a_1 \cdot (b_1 c_1), a_1 \cdot (b_1 c_2 + b_2 c_1) + a_2 \cdot (b_1 c_1)) \\ &= (a_1, a_2) \otimes (b_1 c_1, b_1 c_2 + b_2 c_1) \\ &= (a_1, a_2) \otimes [(b_1, b_2) \otimes (c_1, c_2)] \\ &= \boldsymbol{a} \otimes (\boldsymbol{b} \otimes \boldsymbol{c})\end{aligned}$$

- $\otimes$ distributes over $\oplus$, $\forall a, b, c \in \mathbb{A} = \mathbb{R} \times \mathbb{R}$:

$$\begin{aligned}(\boldsymbol{a} \oplus \boldsymbol{b}) \otimes \boldsymbol{c} &= ((a_1, a_2) \oplus (b_1, b_2)) \otimes (c_1, c_2) \\ &= (a_1 + b_1, a_2 + b_2) \otimes (c_1, c_2) \\ &= ((a_1 + b_1) \cdot c_1, (a_1 + b_1) \cdot c_2 + (a_2 + b_2) \cdot c_1) \\ &= (a_1 c_1 + b_1 c_1, (a_1 c_2 + a_2 c_1) + (b_1 c_2 + b_2 c_1)) \\ &= (a_1 c_1, a_1 c_2 + a_2 c_1) \oplus (b_1 c_1, b_1 c_2 + b_2 c_1) \\ &= ((a_1, a_2) \otimes (c_1, c_2)) \oplus ((b_1, b_2) \otimes (c_1, c_2)) \\ &= (\boldsymbol{a} \otimes \boldsymbol{c}) \oplus (\boldsymbol{b} \otimes \boldsymbol{c})\end{aligned}$$

$$\begin{aligned}\boldsymbol{c} \otimes (\boldsymbol{a} \oplus \boldsymbol{b}) &= (c_1, c_2) \otimes ((a_1, a_2) \oplus (b_1, b_2) \\ &= (c_1, c_2) \otimes (a_1 + b_1, a_2 + b_2) \\ &= (c_1 \cdot (a_1 + b_1), c_2 \cdot (a_1 + b_1) + c_1 \cdot (a_2 + b_2)) \\ &= (c_1 a_1 + c_1 b_1, (c_2 a_1 + c_2 b_1) + (c_1 a_2 + c_1 b_2)) \\ &= (c_1 a_1, c_2 a_1 + c_1 a_2) \oplus (c_1 b_1, c_1 b_2 + c_2 b_1) \\ &= ((c_1, c_2) \otimes (a_1, a_2)) \oplus ((c_1, c_2) \otimes (b_1, b_2)) \\ &= (\boldsymbol{c} \otimes \boldsymbol{a}) \oplus (\boldsymbol{c} \otimes \boldsymbol{b})\end{aligned}$$

- $\mathbf{0}$ is the annihilator for $\otimes$:

$$\begin{aligned}\mathbf{0} \otimes a &= (0, 0) \otimes (a_1, a_2) \\ &= (0 \cdot a_1, 0 \cdot a_2 + 0 \cdot a_1) \\ &= (0, 0 + 0) \\ &= (0, 0) \\ &= \mathbf{0}\end{aligned} \qquad \begin{aligned}a \otimes \mathbf{0} &= (a_1, a_2) \otimes (0, 0) \\ &= (a_1 \cdot 0, a_1 \cdot 0 + a_2 \cdot 0) \\ &= (0, 0 + 0) \\ &= (0, 0) \\ &= \mathbf{0}\end{aligned}$$

b) How I prove this point is by construction, using without the loss of generality the forward algorithm (the same holds for the backwards algorithm). I start with the initial set of values for the vector $\beta(t)$, starting from the first word in the sequence, for all possible taggings. The initial values were chosen as the neutral element for $\otimes$ operation of the semiring. Then I apply the forward algorithm updates of the vector β for subsequent steps, using the semiring's operations $\otimes$ and $\oplus$. I perform the first two updates, which are enough to deduce the general update rule, and finally derive the final value of the β vector, which contains exactly the normalizer $Z(w)$ and the unnormalized entropy $H_U(T_w)$. For clarity reasons the θ subscript has been omitted from score_θ.

Initialization of the forward algorithm:

$$\beta(t_1) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

First update:

$$\begin{aligned}
\beta(t_2) &= \sum_{t_1 \in T} \begin{pmatrix} \exp(\text{score}(t_1, t_2)) \\ -\exp(\text{score}(t_1, t_2))\text{score}(t_1, t_2) \end{pmatrix} \otimes \beta(t_1) \\
&= \sum_{t_1 \in T} \begin{pmatrix} \exp(\text{score}(t_1, t_2)) \\ -\exp(\text{score}(t_1, t_2))\text{score}(t_1, t_2) \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
&= \sum_{t_1 \in T} \begin{pmatrix} \exp(\text{score}(t_1, t_2)) \\ -\exp(\text{score}(t_1, t_2))\text{score}(t_1, t_2) \end{pmatrix} \\
&= \begin{pmatrix} \sum_{t_1 \in T} \exp(\text{score}(t_1, t_2)) \\ \sum_{t_1 \in T} -\exp(\text{score}(t_1, t_2))\text{score}(t_1, t_2) \end{pmatrix}
\end{aligned}$$

Second update:

$$\begin{aligned}
\beta(t_3) &= \sum_{t_2 \in T} \begin{pmatrix} \exp(\text{score}(t_2, t_3)) \\ -\exp(\text{score}(t_2, t_3))\text{score}(t_2, t_3) \end{pmatrix} \otimes \beta(t_2) \\
&= \sum_{t_2 \in T} \begin{pmatrix} \exp(\text{score}(t_2, t_3)) \\ -\exp(\text{score}(t_2, t_3))\text{score}(t_2, t_3) \end{pmatrix} \otimes \begin{pmatrix} \sum_{t_1 \in T} \exp(\text{score}(t_1, t_2)) \\ \sum_{t_1 \in T} -\exp(\text{score}(t_1, t_2))\text{score}(t_1, t_2) \end{pmatrix} \\
&= \sum_{t_2 \in T} \begin{pmatrix} \exp(\text{score}(t_2, t_3)) \sum_{t_1 \in T} \exp(\text{score}(t_1, t_2)) \\ e^{\text{score}(t_2,t_3)} \sum_{t_1} e^{\text{score}(t_1,t_2)}\text{score}(t_1, t_2) - e^{\text{score}(t_2,t_3)}\text{score}(t_2, t_3) \sum_{t_1} e^{\text{score}(t_1,t_2)} \end{pmatrix} \\
&= \sum_{t_2 \in T} \begin{pmatrix} \sum_{t_1 \in T} \exp(\text{score}(t_1, t_2)) \exp(\text{score}(t_2, t_3)) \\ -\sum_{t_1} e^{\text{score}(t_2,t_3)} e^{\text{score}(t_1,t_2)}\text{score}(t_1, t_2) - \sum_{t_1} e^{\text{score}(t_1,t_2)} e^{\text{score}(t_2,t_3)}\text{score}(t_2, t_3) \end{pmatrix} \\
&= \begin{pmatrix} \sum_{t_2} \sum_{t_1} \exp(\text{score}(t_1, t_2) + \text{score}(t_2, t_3)) \\ -\sum_{t_2} \sum_{t_1} e^{\text{score}(t_2,t_3)} e^{\text{score}(t_1,t_2)}\text{score}(t_1, t_2) - \sum_{t_2} \sum_{t_1} e^{\text{score}(t_1,t_2)} e^{\text{score}(t_2,t_3)}\text{score}(t_2, t_3) \end{pmatrix} \\
&= \begin{pmatrix} \sum_{t_2} \sum_{t_1} \exp(\text{score}(t_1, t_2) + \text{score}(t_2, t_3)) \\ -\sum_{t_2} \sum_{t_1} e^{\text{score}(t_1,t_2)+\text{score}(t_2,t_3)}(\text{score}(t_1, t_2) + \text{score}(t_2, t_3)) \end{pmatrix}
\end{aligned}$$

Final value and conclusion of the proof:

$$\begin{aligned}\beta(t_N) &= \begin{pmatrix} \sum_{t_1,\ldots,t_{N-1}} \exp(\sum_{n=1}^{N} \text{score}(t_{n-1}, t_n)) \\ -\sum_{t_1,\ldots,t_{N-1}} \exp(\sum_{n=1}^{N} \text{score}(t_{n-1}, t_n))(\sum_{n=1}^{N} \text{score}(t_{n-1}, t_n)) \end{pmatrix} \\ &= \begin{pmatrix} \sum_{t \in T} \exp(\text{score}(t, w)) \\ -\sum_{t \in T} \exp(\text{score}(t, w))\text{score}(t, w) \end{pmatrix} \\ &= \boxed{\begin{pmatrix} Z(w) \\ H_U(T_w) \end{pmatrix}}\end{aligned}$$

c) Equation 1 is the one that needs to be proven.

$$H(T_w) = Z(w)^{-1} H_U(T) + \log Z(w) \tag{1}$$

Equation 2 is the definition of the entropy of the conditional distribution over taggings t, given a sentence w, $p(t|w)$.

$$H(T_w) = -\sum_{t \in T^N} p(t|w) \log p(t|w) \tag{2}$$

Probability distribution over taggings, given a sentence is represented by the Conditional Random Field model in equation 3.

$$p(t|w) = \frac{\exp(\text{score}(w, t))}{\sum_{t' \in T^N} \exp(\text{score}(w, t'))} = Z(w)^{-1} \exp(\text{score}(w, t)) \tag{3}$$

Plugging the expression for the probability distribution from 3 into the defining expression for entropy 2, one produces the following sequence of expressions:

$$H(T_w) = -\sum_{t \in T^N} \frac{\exp(\text{score}(w, t))}{Z(w)} \log \left(\frac{\exp(\text{score}(w, t))}{Z(w)} \right) \tag{4}$$

$$= -\sum_{t \in T^N} \frac{\exp(\text{score}(w, t))}{Z(w)} (\text{score}(w, t) - \log(Z(w))) \tag{5}$$

$$= -\frac{1}{Z(w)} \sum_{t \in T^N} \exp(\text{score}(w, t))\text{score}(w, t) + \underbrace{\left(\sum_{t \in T^N} \frac{\exp(\text{score}(w, t))}{Z(w)} \right)}_{=\sum_{t \in T^N} p(t|w) = 1} \log Z(w) \tag{6}$$

$$= \boxed{Z^{-1}(w) H_U(T_w) + \log Z(w)} \tag{7}$$

Noting that the left summand in expression 6 is exactly the unnormalized entropy, while the right summand is the product of $\log Z(w)$ and the sum of the distribution $p(t|w)$ over its whole domain, which is equal to 1.

d) Both $Z(w)$ and $H_U(T_w)$ are computed by the forward algorithm in the semiring defined in Definition 1, $< \mathbb{R} \times \mathbb{R}, \oplus, \otimes, \mathbf{0}, \mathbf{1} >$. Since the forward algorithm has computational complexity $O(N \cdot |\mathcal{T}|^2)$, and computing $H(T_w)$ does not require any more

computations, apart from using equation 7, we can conclude that the computational complexity of computing $H(T_w)$ is the same: $O(N \cdot |\mathcal{T}|^2)$. The rationale for the forward algorithm having this complexity is the following: In each step $n = 1, 2, \ldots, N$, one computes, for each tag of the current step, the sum over all tags of the previous step. This involves computing the value $\beta_n(t), \quad \forall t \in \mathcal{T}, \quad \forall n \in 1, 2, \ldots, N$. So, for each layer in the graph, that corresponds to each word in the sentence, one computes the β value in $O(|\mathcal{T}|)$, and does so $|\mathcal{T}|$ times. This means that computing the whole β layer in step n takes $O(|\mathcal{T}|^2)$. This is done for each of N words in the sentence, so the final computational complexity is $\boxed{O(N \cdot |\mathcal{T}|^2)}$.

Question 2

a) First tagging to be popped from the queue will correspond to the path through the graph with the largest score, since Dijkstra's algorithm lifted in the non-positive version of the arctic semiring finds the longest path, i.e. the path with the (non-positive) score that is largest, i.e. closest to zero. Hence, if the scores are negative, running Dijkstra's algorithm to find the path of maximum score actually produces the highest scoring path. Early-stopping does not change this fact. Early stopping will only stop after popping the last node (meaning: node from the last layer, one corresponding to the last word of the sentence) of the highest-scoring path, before finishing the loop of the algorithm, whereas not stopping early will only lead to updating the values and subsequent popping of nodes of all the suboptimal paths from the queue and explicitly assigning their values to the γ matrix. This means that not stopping early esentially means visiting all of the nodes, much like Viterbi's algorithm. Popping the rest of the suboptimal paths after the best node $< n, t >$, for $n = |\boldsymbol{w}|$ does not make any difference to the solution, except that it introduces additional unnecessary computations that can be avoided. If those nodes had better scores, they would have been popped earlier.

   Once a node (n,t) is popped from the queue, it is guaranteed that the score of the path that starts from the first node (0, BOT), and ends with node (n,t) is the best-scoring path ending with the node (n,t). In this context, the word 'best' may refer to either the largest or the smallest score, depending on the semiring that the algorithm is performed in (for example, running Dijkstra's in the tropical semiring ensures that the score of the path from the starting node to the current node is the smallest possible, while doing the same in the arctic semiring would result in the largest score). This is guaranteed by the Dijkstra's algorithm in general.

   Suppose that a node (N, t_N) corresponding to the tag of the last (N^{th}) word is popped from the queue, suppose that its predecessor is node $(N-1, t_{N-1})$ and suppose that it's score is not optimal. This implies that there exists a node $(N-1, t), \; n < N$, such that the path from it to (N, t_N) has a better score. If the node $(N-1, t)$ was popped, then the nodes from the last layer, layer N, with their corresponding scores would have been added to the queue, and if they were better than the one coming from $(N-1, t_{N-1})$ they would have been popped from the priority queue earlier, which is a contradiction with the aforementioned assumptions. In another case, if node $(N-1, t)$ was not popped from the queue, and $(N-1, t_{N-1})$ was popped, this

implies that the score for $(N-1, t_{N-1})$ was better than for $(N-1, t)$, which again represents a contradiction with the assumption that $(N-1, t)$ had a better score. Taking all of the above into account, it can be concluded that the first score of the tagging to be popped from the queue corresponds to the score of the best tagging for the sentence.

b) Viterbi's forward algorithm fills out the values of γ the following way:

$$2 \leq n \leq N,\ \forall t_n \in T : \gamma[t_n, n] \longleftarrow \max_{t_{n-1} \in T} \exp\{\text{score}(< t_{n-1}, t_n >)\} + \gamma[t_{n-1}, n-1]$$

It can be noted that Viterbi's algorithm computes, starting from $\gamma[t_1, 1] = \exp\{\text{score}(< \text{BOT}, t_1 >)\}$, all of the entries of γ in an ordered manner, starting from the initial node of the graph, and continuing through the levels n sequentially. It computes all of the entries in γ, i.e. the algorithm terminates only when all of the values of γ are computed.

Dijkstra's algorithm fills out the $\gamma[n, t]$ values in the following way:

$$\gamma[n+1, t'] = \text{score}(< t, t' >) + \gamma[n, t]$$

However, the score $\gamma[n+1, t']$ would correspond to $\max_{t_{n-1} \in T} \exp\{\text{score}(< t_{n-1}, t_n >)\} + \gamma[t_{n-1}, n-1]$ due to the priority queue popping the node $(n+1, t')$ with the highest score, and node $(n-1, t')$ having the highest score for its position. Compared to Viterbi's algorithm, Dijkstra's algorithm does not have a fixed schedule of score computation for the nodes $\gamma[n, t]$. Rather, there is a queue that determines the ordering of the nodes whose values are to be updated. The queue uses the $\oplus$ operation to order its elements, which in the case of the Arctic semiring boils down to the max operation, meaning that the first element to be popped out of the queue in each iteration is the one with the highest score. So, to reiterate, the element (n, t) that is popped in the current iteration has the best possible score out of all the elements in the queue. Through this element, and its score, we update the scores of the neighbouring unvisited nodes in the next layer (i.e. nodes corresponding to tags of the next word in the sentence). Finally, with early stopping, when an element of the last, i.e. N-th layer of the graph is popped, it is guaranteed that the best possible tagging has been found. This means that all of the other elements have worse scores, which in turn means that the rest of the nodes in the last layer cannot have better scores than the one popped, as scores add up.

$$\begin{aligned}\gamma[N, t] &= \max_{t_{N-1}}(\exp\{\text{score}(t_{N-1}, t_N)\} + \gamma[N-1, t_{N-1}]) \\ &= \max_{t_1, \ldots, t_N}(\exp(\text{score}(t_1, \ldots, t_N))\end{aligned}$$

With early stopping, some of the $\gamma[n, t]$ values might end up unknown, since the Dijkstra's algorithm would terminate before visiting every node of the graph. However, in the case of the version without early stopping, the algorithm runs until the queue is empty. In other words, algorithm runs until all of the nodes are visited, and their values updated to the best (according to the partial ordering of the semiring that the algorithm is operated in) values possible for each node. In that sense, the γ values of both the Viterbi's and Dijkstra's algorithm compute the same values, provided that Dijkstra's is run until the queue is empty, without early stopping.

c) In the worst case scenario, the priority queue will have contained all of the nodes of the graph. Meaning that there would have been $|N \cdot T|$ nodes in the queue. As the upper bound, Dijkstra's without early stopping is analyzed. This version would correspond to the worst case scenario of Dijkstra's algorithm with early stopping, too. In this version, all of the elements in the queue must be popped for the algorithm to terminate. Assuming a heap implementation of the priority queue, pushing each of $|N \cdot T|$ items takes $O(\log(|N \cdot T|))$ time, while popping takes $O(1)$. Each popped element has $|T|$ successor nodes that need to be pushed to the queue, which takes $O(\log(|N \cdot T|))$ time for each one of the successor nodes. Finally, to sum up, for each of $N \cdot |T|$ nodes, there is a pop operation done in $O(1)$ and $|T|$ push operations at the cost of $O(\log(N \cdot |T|))$ per push, which means that the total complexity is $O(N \cdot |T| + N \cdot |T|^2 \log(N \cdot |T|)) = \boxed{O\left(N \cdot |T|^2 \log(N \cdot |T|)\right)}$.

d) The $\oplus$ operation is the one that dictates the key updates in the priority queue. With the $\oplus_{\log}$, the update becomes:

$$\begin{aligned}\text{PriorityQueue}[< n, t >] &:= \text{PriorityQueue}[< n, t >] \oplus_{\log} x \\ &= -\log\left(\exp(-x) + \exp\left(-\text{PriorityQueue}[< n, t >]\right)\right)\end{aligned}$$

Here the value x is an alternative value that the node $< n, t >$ can have. Updating the value where $\oplus = \min$ would mean that the new value of the node $< n, t >$ is the minimum of the old score PriorityQueue$[< n, t >]$ and the new score x. This guarantees proper functioning of Dijkstra's algorithm in the case of a semiring with non-positive values (the "modified" arctic semiring). However, in the case where $\oplus_{\log}$ is used, things are different. First, note that the result of applying the $\oplus_{\log}$ operation is always smaller than the min function value, for any two inputs, i.e. it holds:

$$\forall(x,\ y) \in \mathbb{R} \times \mathbb{R} : \quad x \oplus_{\log} y \leq \min(x, y)$$

The min function is its upper bound, and the $\oplus_{\log}$ is in a sense a "soft-min" function, since it is differentiable, unlike min. The fact that the set of the proposed semiring is all real numbers, with the addition of ∞ and $-\infty$, and the fact that the $\oplus_{\log}$ behaves as a minimum in some sense, means that Dijkstra's algorithm, much like when used with the min function and negative values of nodes, is not guaranteed to find the best possible solution. This may be possible in the case where the set of values is limited from below with 0.

As an example of why negative weights can negatively affect the outcome of Dijkstra's algorithm, we can look at a graph shown in 1. First node to be popped into the queue is (BOT, 0), then we add nodes 2, with score 1, and node 3 with score 3. Since node 2 has lower score, it's popped first, and nodes 4, with score 2 and node 5 with score 2 are added to the queue. Node 4 is popped, as it's score is lowest in the queue. As node 4 is in the last layer, the algorithm stops. However, this is not the minimum score for the path, as it's obvious from the graph that the minimum path would have score -6, and would necessarily pass through node 3. However, since node 3 has initially a worse score than node 2, it is never explored, under the assumption that the weights are non-negative and can only be worse than passing through node 2. Same would happen with $\oplus_{\log}$

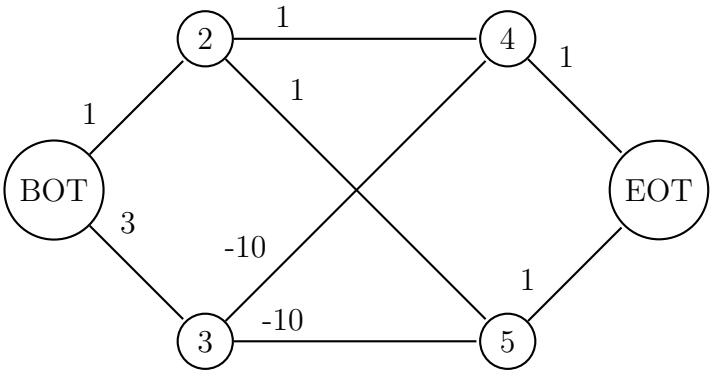

Figure 1: Counterexample: What happens when negative weights are used when $\oplus$ is min

e) Running Dijkstra with $\oplus$ = min over negative values, or $\oplus$ = max over negative values has the consequence that it is not guaranteed to produce the path with the best score. This happens because in the case of negative weights and $\oplus$ = max, pushing new elements onto the stack, one cannot guarantee that the currently popped, i.e. visited element has the score that corresponds to the best path to that element. This is explained in the previous section, and illustrated in figure 1. Analogous holds for $\oplus$ = max. There might be better paths that are unexplored, whose scores are negative, which would formally produce a better path. Limiting the set from above (below in case of $\oplus$ = min) with the neutral element for $\oplus$ would allow us to guarantee in each step that the score of the path from start node to any current node is the best possible score to that node. We know the values of all other paths to that node, and we know that the scores cannot improve over the best one, if the weights of the edges are non-negative.

   Running Dijkstra's using max as $\oplus$ and non-negative weights means that as the algorithm progresses through the paths, its will greedily select the nodes with the highest score, event though they are not guaranteed to be the best possible option. In a sense, there must be a mechanism of penalization when choosing nodes, such that the actual best ones are chosen in each step.

   To sum up, any semiring $(A, \min, +, \infty, 0)$, where the set A is limited from below, i.e. $A = \{x | x \geq 0\} \cup \{\infty\}$ would guarantee the correctness of Dijkstra's algorithm.

Question 3

`https://colab.research.google.com/drive/1c0OLag5EsnvKgw3Cev_uBI5D5Fm_oxWL?usp=sharing`

Note:

I have modified the provided code to make it possible to change the device the algorithms are ran on. Modifications consist of sending the models and data to the GPU when training, and turning off the use of deterministic algorithms during training (this was required by pytorch). The other parts of the code are not changed, as they are still ran on CPU, to

guarantee reproducibility. Using GPU to train the models has cut the training time tenfold - from about 60 minutes per epoch to about 6 minutes per epoch. The total training time for the 4 training cells with 3 epochs each was about an hour.

e)

The runtimes of Viterbi's and Dijkstra's differ significantly, which is in line with the expectations, given the computational complexities given in the answers to Question 2. Viterbi's is significantly faster. Moreover, Viterbi's algorithm profits off of the vectorization of tensors, which only further increases its speed. Dijkstra's however, cannot be parallelized, as it visits and computes values for one node at a time.

f)

During training, the accuracy increases, until it reaches the final per-tag accuracy of about 90.5% on the held-out dataset, after the third epoch. The training results are shown on figure 2.

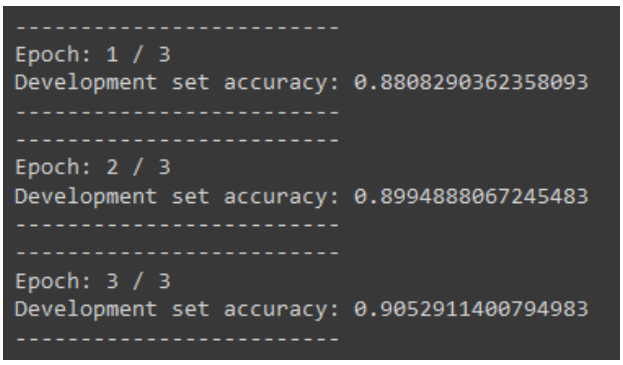

Figure 2: Results during the training over three epochs or the final CRF model.

g)

Adding entropy of the final model's output as an additional term to the overall objective function has a regularizing effect. With higher values of β, the entropy term has more and more influence over the training, and thus the final values of parameters. Adding entropy to the objective function makes the final output of the probabilistic model have less pronounced and concentrated peaks. The model stops being overconfident in its classification of the samples provided to it. This means that overconfidence, i.e. low-entropy behaviour apparent in the concentrated peaks over labels is penalized, and high-entropy outputs are encouraged. The probability is more spread out over the labels than it would have been without the entropy term. How the training parameter β affects this behaviour in terms of the development set accuracy is shown in figure 3.

All of the results are relatively close to each other, with the version where $\beta = 10$ achieves the highest accuracy out of the three, at 92.13%. However, since I trained the model

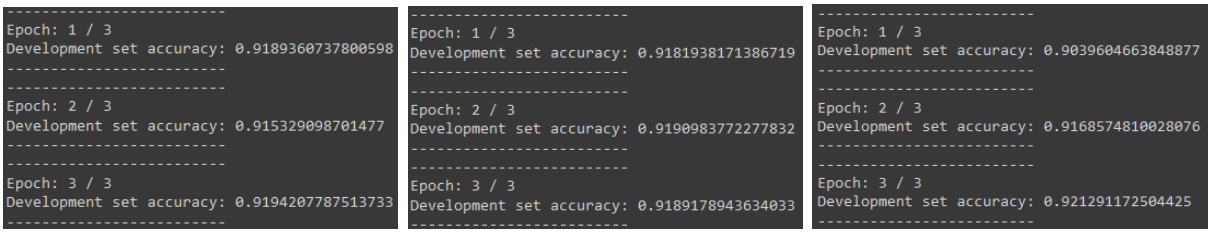

Figure 3: Per-tag accuracies on the development set for the CRF model trained with three values of $\beta \in \{0.1, 1, 10\}$, from left to right, respectively.

on GPU, which required the use of deterministic algorithms to be turned off, the above-mentioned results may not be completely representative.