Natural Language Processing: Assignment 2

A Detailed Conceptual Explanation

Original Author:Mohammad Taha Majlesi (810101504)

Fall 2022

Abstract

This document provides a detailed, pedagogical walkthrough of the concepts presented in Assignment 2 for the Natural Language Processing course. It expands on the original solutions, offering deeper explanations of the underlying algebraic structures (monoids and semirings), the algorithms that operate on them (Forward, Viterbi, Dijkstra), and their practical application in training and analyzing Conditional Random Field (CRF) models. Each section is designed to build a strong conceptual understanding from the ground up.

Contents

1 Algebraic Foundations for NLP

1.1 Introduction to Algebraic Structures

In computer science, and particularly in Natural Language Processing, we often encounter problems that involve finding the "best" path or aggregating information over a vast number of possibilities (e.g., all possible tag sequences for a sentence). Algebraic structures like **monoids** and **semirings** provide a powerful and abstract framework to describe these computations.

By defining our problem in terms of a semiring (a set with "addition" and "multiplication" operations), we can use a single, general algorithm (like the forward algorithm) to solve different problems simply by swapping out the semiring.

- To find the most probable path (Viterbi), we use the **Tropical Semiring**.
- To sum the probabilities of all paths (Forward algorithm), we use the **Probability Semiring**.
- To compute expected values for model training, we use the **Expectation Semiring**, which is the focus of this question.

1.2 Part 1: The Commutative Monoid $< \mathbb{R} \times \mathbb{R}, \oplus, \mathbf{0} >$

This part of the question asks us to prove that a specific structure is a **commutative monoid**. Let's break down what that means.

Definition 1.1 (Commutative Monoid)**.** *A structure $(S, \oplus, e)$ is a **commutative monoid** if it satisfies the following five properties:*

1. ***Closure:** For any two elements $a, b \in S$, the result $a \oplus b$ is also in S.*
2. ***Identity Element:** There exists a special element $e \in S$ such that for any element $a \in S$, $a \oplus e = e \oplus a = a$.*
3. ***Associativity:** For any elements $a, b, c \in S$, the equation $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ holds. The grouping of operations does not matter.*
4. ***Commutativity:** For any two elements $a, b \in S$, the equation $a \oplus b = b \oplus a$ holds. The order of operands does not matter.*

Here, our set is $S = \mathbb{R} \times \mathbb{R}$ (pairs of real numbers), our operation is $\oplus$, and our identity element is $\mathbf{0} = (0, 0)$. Closure is given since adding real numbers results in a real number. Let's verify the other properties.

1.2.1 Verifying the Identity Element

The identity element, when combined with any other element, should leave it unchanged. Here, we test $\mathbf{0} = (0, 0)$. Let $\mathbf{a} = (a_1, a_2)$ be any element in $\mathbb{R} \times \mathbb{R}$.

$$\begin{aligned}\mathbf{a} \oplus \mathbf{0} &= (a_1, a_2) \oplus (0, 0) \\ &= (a_1 + 0, a_2 + 0) \quad \text{(by definition of } \oplus\text{)} \\ &= (a_1, a_2) = \mathbf{a}\end{aligned}$$

Since standard addition is commutative, $\mathbf{0} \oplus \mathbf{a}$ also equals $\mathbf{a}$. Thus, $\mathbf{0}$ is the identity element.

1.2.2 Verifying Associativity

Associativity ensures that we can sum a long sequence of elements without ambiguity. Let $\mathbf{a} = (a_1, a_2)$, $\mathbf{b} = (b_1, b_2)$, and $\mathbf{c} = (c_1, c_2)$.

$$\begin{aligned}(\mathbf{a} \oplus \mathbf{b}) \oplus \mathbf{c} &= [(a_1, a_2) \oplus (b_1, b_2)] \oplus (c_1, c_2) \\ &= (a_1 + b_1, a_2 + b_2) \oplus (c_1, c_2) \\ &= ((a_1 + b_1) + c_1, (a_2 + b_2) + c_2) \\ &= (a_1 + (b_1 + c_1), a_2 + (b_2 + c_2)) \quad \text{(associativity of real addition)} \\ &= (a_1, a_2) \oplus (b_1 + c_1, b_2 + c_2) \\ &= \mathbf{a} \oplus (\mathbf{b} \oplus \mathbf{c})\end{aligned}$$

The property holds.

1.2.3 Verifying Commutativity

Commutativity ensures the order of summation does not matter.

$$\begin{aligned}\mathbf{a} \oplus \mathbf{b} &= (a_1, a_2) \oplus (b_1, b_2) \\ &= (a_1 + b_1, a_2 + b_2) \\ &= (b_1 + a_1, b_2 + a_2) \quad \text{(commutativity of real addition)} \\ &= (b_1, b_2) \oplus (a_1, a_2) = \mathbf{b} \oplus \mathbf{a}\end{aligned}$$

The property holds. Since all conditions are met, $< \mathbb{R} \times \mathbb{R}, \oplus, \mathbf{0} >$ is a commutative monoid.

1.3 Part 2: The Monoid $< \mathbb{R} \times \mathbb{R}, \otimes, \mathbf{1} >$

This structure involves a different, more complex operation $\otimes$. For this to be a **monoid**, it only needs to satisfy closure, identity, and associativity (not commutativity). The proposed identity element is $\mathbf{1} = (1, 0)$.

1.3.1 Verifying the Identity Element

Let's test if $\mathbf{1} = (1, 0)$ acts as an identity for the $\otimes$ operation.

$$\begin{aligned}\mathbf{a} \otimes \mathbf{1} &= (a_1, a_2) \otimes (1, 0) \\ &= (a_1 \cdot 1, a_1 \cdot 0 + a_2 \cdot 1) \quad \text{(by definition of } \otimes\text{)} \\ &= (a_1, 0 + a_2) = (a_1, a_2) = \mathbf{a}\end{aligned}$$

It works from the right. Let's check from the left:

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

It works from both sides. Thus, $\mathbf{1} = (1, 0)$ is the identity element.

1.3.2 Verifying Associativity

This proof is more involved and demonstrates the non-trivial nature of the operation.

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

Now, we group the operations differently:

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

The resulting expressions are identical. Therefore, the $\otimes$ operation is associative, and the structure is a monoid.

1.4 Part 3: The Expectation Semiring

A semiring combines two monoid structures.

Definition 1.2 (Semiring)**.** *A structure $(S, \oplus, \otimes, e, u)$ is a **semiring** if:*

1. *$(S, \oplus, e)$ is a commutative monoid (the "additive" structure).*
2. *$(S, \otimes, u)$ is a monoid (the "multiplicative" structure).*
3. ***Distributivity:** $\otimes$ distributes over $\oplus$. For all $a, b, c \in S$:*

- $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *(Left distributivity)*
- $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$ *(Right distributivity)*

4. ***Annihilation:*** *The additive identity e is an annihilator for $\otimes$. For all $a \in S$, $a \otimes e = e \otimes a = e$.*

We have already proven the two monoid properties. Now we verify distributivity and annihilation for $< \mathbb{R} \times \mathbb{R}, \oplus, \otimes, \mathbf{0}, \mathbf{1} >$.

1.4.1 Verifying Distributivity

We must show that multiplication distributes over addition. Let's prove right distributivity:

$$\begin{aligned}(\mathbf{a} \oplus \mathbf{b}) \otimes \mathbf{c} &= (a_1 + b_1, a_2 + b_2) \otimes (c_1, c_2) \\ &= ((a_1 + b_1)c_1, (a_1 + b_1)c_2 + (a_2 + b_2)c_1) \\ &= (a_1c_1 + b_1c_1, a_1c_2 + b_1c_2 + a_2c_1 + b_2c_1)\end{aligned}$$

Meanwhile:

$$\begin{aligned}(\mathbf{a} \otimes \mathbf{c}) \oplus (\mathbf{b} \otimes \mathbf{c}) &= (a_1c_1, a_1c_2 + a_2c_1) \oplus (b_1c_1, b_1c_2 + b_2c_1) \\ &= (a_1c_1 + b_1c_1, (a_1c_2 + a_2c_1) + (b_1c_2 + b_2c_1)) \\ &= (a_1c_1 + b_1c_1, a_1c_2 + b_1c_2 + a_2c_1 + b_2c_1)\end{aligned}$$

The results are equal. A similar proof holds for left distributivity.

1.4.2 Verifying Annihilation

We must show that multiplying by the additive identity $\mathbf{0} = (0, 0)$ results in $\mathbf{0}$.

$$\mathbf{a} \otimes \mathbf{0} = (a_1, a_2) \otimes (0, 0) = (a_1 \cdot 0, a_1 \cdot 0 + a_2 \cdot 0) = (0, 0) = \mathbf{0}$$
$$\mathbf{0} \otimes \mathbf{a} = (0, 0) \otimes (a_1, a_2) = (0 \cdot a_1, 0 \cdot a_2 + 0 \cdot a_1) = (0, 0) = \mathbf{0}$$

This property holds. All conditions are met, confirming this structure is a semiring.

2 Algorithms on Semirings

2.1 The Forward Algorithm in the Expectation Semiring

The forward algorithm is a cornerstone of sequence models. It efficiently computes a value, $\beta(t)$, representing the total "score" of all paths from the start of a sequence to a given state at time t. What "score" means is defined by the semiring.

In the Expectation Semiring, a score is a pair (x, y), where:

- x corresponds to a probability (or more generally, an exponential score).
- y corresponds to the score multiplied by some value of interest.

When we use $\oplus$ (component-wise addition), we are summing these values over different paths. When we use $\otimes$, we are combining scores along a single path.

2.1.1 Initialization and Recursion

- **Initialization:** The algorithm starts with $\beta(t_1) = \mathbf{1} = (1, 0)$. This represents a path of length zero, which has a total probability of 1 and an accumulated value of 0.
- **First Update:** To compute $\beta(t_2)$, we sum over all possible starting tags t_1. The score of a transition from t_1 to t_2 is represented in the semiring by the pair $(e^{\text{score}(t_1,t_2)}, e^{\text{score}(t_1,t_2)}\text{score}(t_1, t_2))$. The update rule is:

$$\beta(t_2) = \bigoplus_{t_1 \in T} (\text{semiring\_score}(t_1, t_2) \otimes \beta(t_1))$$

  This calculation combines the path score up to t_1 with the transition score to t_2 (using $\otimes$) and then sums these possibilities (using $\oplus$).

2.1.2 Final Value and Interpretation

After processing the entire sentence of length N, the final value is obtained by summing the β values at the last step: $\beta_{final} = \bigoplus_{t_N \in T} \beta(t_N)$. This final vector holds two crucial quantities:

$$\beta_{final} = \begin{pmatrix} \sum_t e^{\text{score}(t,w)} \\ \sum_t e^{\text{score}(t,w)}\text{score}(t, w) \end{pmatrix} = \begin{pmatrix} Z(w) \\ H_U(T_w) \end{pmatrix}$$

- $Z(w)$: The **partition function**. It is the sum of the exponentiated scores over *all possible tag sequences*. This value is essential for normalizing scores into a valid probability distribution.
- $H_U(T_w)$: The **unnormalized entropy** (or more accurately, unnormalized expected score). This is the key component needed to calculate the entropy of the model's posterior distribution.

The beauty of the semiring framework is that a single pass of the forward algorithm computes both of these values simultaneously.

2.2 Entropy, Complexity, and Model Analysis

2.2.1 Connecting the Forward Pass to Entropy

The entropy of the model's output distribution $p(t|w)$ is a measure of its uncertainty. It is formally defined as $H(T_w) = -\sum_t p(t|w) \log p(t|w)$. In a CRF, $p(t|w) = \exp(\text{score}(t, w))/Z(w)$. By substituting this into the entropy formula, we get:

$$\begin{aligned} H(T_w) &= -\sum_t \frac{e^{\text{score}}}{Z(w)} \log\left(\frac{e^{\text{score}}}{Z(w)}\right) \\ &= -\frac{1}{Z(w)} \sum_t e^{\text{score}}(\text{score} - \log Z(w)) \\ &= -\frac{1}{Z(w)} \left(\sum_t e^{\text{score}}\text{score} - \log Z(w) \sum_t e^{\text{score}}\right) \\ &= -\frac{1}{Z(w)}(H_U(T_w) - Z(w) \log Z(w)) \\ &= Z(w)^{-1} H_U(T_w) + \log Z(w) \end{aligned}$$

This elegant result shows that entropy can be calculated in constant time once $Z(w)$ and $H_U(T_w)$ are known from the forward pass.

2.2.2 Computational Complexity

The forward algorithm's complexity is $O(N \cdot |T|^2)$, where N is the sentence length and $|T|$ is the tag set size. This arises because:

- We iterate through each of the N words in the sentence.
- For each word, we compute a β value for each of its $|T|$ possible tags.
- To compute each of these, we must sum the contributions from all $|T|$ tags of the previous word.

This results in a total of $N \times |T| \times |T|$ core operations.

2.3 Viterbi vs. Dijkstra's Algorithm

Both Viterbi and Dijkstra's algorithm can find the best-scoring path in a graph.

- **Viterbi's Algorithm** is a dynamic programming algorithm tailored for sequence models. It explores the graph in a fixed, layer-by-layer (or word-by-word) order. It computes the best path to *every* node at step n before considering any node at step $n + 1$. This structured approach makes it extremely efficient and easy to vectorize. It is equivalent to running the forward algorithm on the **Tropical Semiring** $(\mathbb{R} \cup \{-\infty\}, \max, +, -\infty, 0)$.

- **Dijkstra's Algorithm** is a general-purpose graph search algorithm. It maintains a priority queue of nodes to visit and always expands the node with the best score found so far, regardless of its position in the sequence. While more general, its less structured exploration and reliance on a priority queue make it harder to optimize for sequence models.

2.3.1 The Problem with Negative Weights

Dijkstra's algorithm's correctness guarantee—that the first time it reaches a node, it has found the shortest path to it—relies on **non-negative edge weights**. If weights can be negative, the greedy choice made by Dijkstra might be wrong.

Consider the example graph. Dijkstra's, seeking a 'max' score (or longest path), would explore the path BOT $\rightarrow$ 2 because its initial score is higher than BOT $\rightarrow$ 3. It would never discover the path through node 3, which ultimately leads to a much better score due to the large positive-valued edges (in a max-path context, this is analogous to large negative weights in a min-path context).

This means that for a semiring to guarantee the correctness of Dijkstra's algorithm, its values must be restricted in a way that prevents this "deceptive" path problem (e.g., by being non-negative for a shortest-path problem).

3 Experimental Analysis of a CRF Model

3.1 Practical Runtimes and Performance

Experiments confirm the theoretical analysis:

- **Viterbi is significantly faster** than Dijkstra's. Its layer-wise computation is perfectly suited for modern hardware, which excels at vectorized operations on tensors.
- **Dijkstra's is slower** because its core logic involves repeatedly pushing to and popping from a priority queue, an inherently sequential process that cannot be easily parallelized.

3.2 Entropy as a Regularization Technique

Definition 3.1 (Regularization)**.** ***Regularization*** *is any technique added to a machine learning model to prevent* ***overfitting****. Overfitting occurs when a model learns the training data too well, including its noise and idiosyncrasies, causing it to perform poorly on new, unseen data. Regularization introduces a penalty for model complexity, encouraging simpler, more generalizable solutions.*

Adding entropy to the objective function is a powerful form of regularization.

- A model that is **overconfident** produces low-entropy (peaky) probability distributions. It puts all its probability mass on a single outcome.
- By adding a term $-\beta H(T_w)$ to the objective function we are maximizing, we reward the model for having higher entropy. This encourages the model to spread its probability mass a bit more, making it less "certain" and more robust.

The hyperparameter β controls the strength of this effect. The experiments show that a model trained with $\beta = 10$ achieved the highest accuracy (92.13%), suggesting that for this task, preventing overconfidence via entropy regularization was beneficial.

3.3 Note on Reproducibility

The experiments were conducted on a GPU. For performance reasons, many deep learning libraries use non-deterministic algorithms on GPUs. This means that running the same code multiple times may produce slightly different results, a crucial factor to consider when evaluating and comparing models.