Lecture 7: The Viterbi algorithm

HMM recap (I)

An HMM is a **generative** probability model of word and hidden state (e.g. tag) sequences.

\[ P(w, t) = P(t) \cdot P(w | t) \]

For a bigram HMM:

\[
P(w = w^{(1)}w^{(2)}...w^{(N)}, t = t^{(1)}t^{(2)}...t^{(N)}) =
\]

\[ P(t^{(1)}) \cdot P(w^{(1)} | t^{(1)}) \cdot P(t^{(2)} | t^{(1)}) \cdot P(w^{(2)} | t^{(2)}) \cdot ... \cdot P(w^{(N)} | t^{(N)}) \]

**Tagging with an HMM** = finding the most likely sequence of hidden states that generated \( w \):

\[ t^* = \text{argmax}_t P(w, t) \]

This search for the best \( t \) is also called **decoding**.

HMM recap (II)

An HMM \( \lambda = (A, B, \pi) \) consists of:

- a **transition** matrix \( A \): defines transition probabilities \( P(q_i | q_j) \)
- an **emission** matrix \( B \): defines emission probabilities \( P(w_i | q_j) \)
- an **initial state** vector \( \pi \): defines an initial state distribution \( P(q_i) \)

In a bigram HMM tagger, each state \( q_i \) corresponds to one POS tag (the tag of the current word): \( q_i = t_i \)

An example HMM

<table>
<thead>
<tr>
<th>Transition Matrix ( A )</th>
<th>Emission Matrix ( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>the</td>
</tr>
<tr>
<td>D</td>
<td>0.8</td>
</tr>
<tr>
<td>N</td>
<td>0.7</td>
</tr>
<tr>
<td>V</td>
<td>0.6</td>
</tr>
<tr>
<td>A</td>
<td>0.8</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial state vector ( \pi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
</tr>
<tr>
<td>π</td>
</tr>
</tbody>
</table>
HMMs as probabilistic automata

An HMM defines
Transition probabilities:
P(t_i | t_j)
Emission probabilities:
P(w_i | t_i)

HMM definition

A HMM \( \lambda = (A, B, \pi) \) consists of

- a set of \( N \) states \( Q = \{q_1, \ldots, q_N\} \)
- an output vocabulary of \( M \) items \( V = \{v_1, \ldots, v_M\} \)
- a set of \( N \) initial states \( Q_i \subseteq Q \) a set of initial (accepting) states
- an initial state distribution vector \( \pi = (\pi_1, \ldots, \pi_N) \)

States = Tag Unigrams

Encoding a trigram model as FSA

Bigram model:
States = Tag Unigrams
Trigram model:
States = Tag Bigrams

How would the automaton for a trigram HMM with transition probabilities \( P(t_i | t_{jtk}) \) look like?

What about unigrams or n-grams?
Trigram HMMs

In a **trigram HMM tagger**, each state $q_i$ corresponds to a POS tag bigram (the tags of the current and preceding word): $q_i = t_jt_k$

Emission probabilities depend only on the current POS tag: States $t_jt_k$ and $t_it_k$ use the same emission probabilities $P(w_i \mid t_k)$

Building an HMM tagger

To build an HMM tagger, we have to:

- Train the model, i.e. estimate its parameters (the transition and emission probabilities)
  - Easy case: we have a corpus labeled with POS tags (supervised learning)
  
- Define and implement a tagging algorithm that finds the best tag sequence $t^*$ for each input sentence $w$:
  
  $$t^* = \arg\max_t P(t)P(w \mid t)$$

Learning an HMM from *labeled* data

We count how often we see $t_jt_i$ and $w_jt_i$ etc. in the data (use relative frequency estimates):

- Transition probabilities:
  
  $$P(t_j \mid t_i) = \frac{C(t_it_j)}{C(t_i)}$$

- Emission probabilities:
  
  $$P(w_j \mid t_i) = \frac{C(w_jt_i)}{C(t_i)}$$

- Initial state probabilities:
  
  $$\pi(t_i) = \frac{C(\text{Tag of first word } = t_i)}{\text{Number of sentences}}$$
Finding the best tag sequence

The number of possible tag sequences is exponential in the length of the input sentence:

Each word can have up to T tags.
There are N words.
There are up to $T^N$ possible tag sequences.

We cannot enumerate all $T^N$ possible tag sequences.

But we can exploit the independence assumptions in the HMM to define an efficient algorithm that returns the tag sequence with the highest probability in linear ($O(N)$) time.

HMM decoding

We observe a sentence $w = w^{(1)} \ldots w^{(N)}$

$w =$ “she promised to back the bill”

We want to use an HMM tagger to find its POS tags $t$

$t^* = \text{argmax}_t P(w, t)$

$= \text{argmax}_t P(t^{(1)}) \cdot P(w^{(1)} | t^{(1)}) \cdot P(t^{(2)} | t^{(1)}) \cdot \ldots \cdot P(w^{(N)} | t^{(N)})$

To do this efficiently, we will use a technique called dynamic programming to exploit the independence assumptions in the HMM.

Dynamic programming

Dynamic programming is a general technique to solve certain complex search problems by memoization

1.) Recursively decompose the large search problem into smaller subproblems that can be solved efficiently
   - There is only a polynomial number of subproblems.

2.) Store (memoize) the solution of each subproblem in a common data structure
   - Processing this data structure takes polynomial time

The Viterbi algorithm

A dynamic programming algorithm which finds the best (=most probable) tag sequence $t^*$ for an input sentence $w$: $t^* = \text{argmax}_t P(w | t)P(t)$

Complexity: linear in the sentence length.
With a bigram HMM, Viterbi runs in $O(T^2N)$ steps for an input sentence with N words and a tag set of T tags.

The independence assumptions of the HMM tell us how to break up the big search problem (find $t^* = \text{argmax}_t P(w | t)P(t)$) into smaller subproblems.

The data structure used to store the solution of these subproblems is called a trellis.
**Bookkeeping: the trellis**

We use a N×T table ("trellis") to keep track of the HMM. The HMM can assign one of the T tags to each of the N words.

**HMM independences**

1. **Emissions** depend only on the current tag:
   \[ \ldots P(w^{(i)} = \text{man} \mid t^{(i)} = \text{NN}) \ldots \]

   We only have to multiply the emission probability \( P(w^{(i)} \mid t_j) \) with the **probability of the best tag sequence** that gets us to \( t^{(i)} = t_j \)

2. **Transition probabilities to the current tag** \( t^{(i)} \) depend only on the previous tag \( t^{(i-1)} \):
   \[ \ldots P(t^{(i)} = \text{NN} \mid t^{(i-1)} = \text{DT}) \ldots \]

   - Assume the **probability of the best tag sequence for the prefix** \( w^{(1)} \ldots w^{(i-1)} \) that ends in the tag \( t^{(i-1)} = t_j \) is known, and stored in a variable \( \max[i-1][j] \).
   - To compute the **probability of the best tag sequence for** \( w^{(1)} \ldots w^{(i)} \) \( w^{(i)} \) that ends in the tags \( t^{(i)} = t_j \), multiply \( \max[i-1][j] \) with \( P(t_k \mid t_j) \) and \( P(w^{(i)} \mid t_k) \)
   - To compute the **probability of the best tag sequence for** \( w^{(1)} \ldots w^{(i)} \) \( w^{(i)} \) that ends in \( t^{(i)} = t_k \), consider all possible tags \( t^{(i-1)} = t_j \) for the preceding word: \( \max[i][k] = \max_j ( \max[i-1][j] P(t_k \mid t_j) P(w^{(i)} \mid t_k) ) \)
HMM independences

3. The current tag also determines the transition probability of the next tag:

\[ P( t(i+1) = \text{VBZ} \mid t(i) = \text{NN} ) \]

We cannot fix the current tag \( t(i) \) based on the probability of getting to \( t(i) \) (and producing \( w(i) \))

We have to wait until we have reached the last word in the sequence. Then, we can trace back to get the best tag sequence for the entire sentence.

Using the trellis to find \( t^* \)

Let \( \text{trellis}[i][j] \) (word \( w(i) \) and tag \( t_j \)) store the probability of the best tag sequence for \( w(1) \ldots w(i) \) that ends in \( t_j \)

\[ \text{trellis}[i][j] = \max P(w(1) \ldots w(i), t(1), \ldots, t(i) = t_j) \]

We can recursively compute \( \text{trellis}[i][j] \) from the entries in the previous column \( \text{trellis}[i-1][j] \)

\[ \text{trellis}[i][j] = P(w(i) \mid t_j) \cdot \max_k ( \text{trellis}[i-1][k] P(t_j \mid t_k) ) \]

At the end of the sentence, we pick the highest scoring entry in the last column of the trellis

Retrieving \( t^* = \arg\max_t P(t, w) \)

By keeping one backpointer from each cell to the cell in the previous column that yields the highest probability, we can retrieve the most likely tag sequence when we’re done.

The Viterbi algorithm

\[
\text{Viterbi( } w_1 \ldots n) \\
\text{ for } t (1 \ldots T) \text{ // INITIALIZATION} \\
\quad \text{trellis}[1][1].viterbi = p_{init}[t] \times p_{emit}[t][w] \\
\text{for } i (2 \ldots n) \text{ // RECURSION} \\
\quad \text{for } t (1 \ldots T) \\
\quad \quad \text{trellis}[i][t].viterbi = 0 \\
\quad \quad \text{for } t' (1 \ldots T) \\
\quad \quad \quad \text{tmp} = \text{trellis}[i-1][t'].viterbi \times p_{trans}[t'][t] \\
\quad \quad \quad \text{if } (\text{tmp} > \text{trellis}[i][t].viterbi) \\
\quad \quad \quad \quad \text{trellis}[i][t].viterbi = \text{tmp} \\
\quad \quad \quad \quad \text{trellis}[i][t].backpointer = t' \\
\quad \quad \quad \text{trellis}[i][t].viterbi \times p_{emit}[t][w] \\
\text{t_max = NULL, vit_max = 0; // FINISH} \\
\text{for } t (1 \ldots T) \\
\quad \text{if } (\text{trellis}[n][t].vit > \text{vit_max}) \{ \text{t_max = t; vit_max = trellis}[n][t].value \} \\
\text{return unpack(n, t_max);}
\]
Unpacking the trellis

```java
unpack(n, t){
    i = n;
    tags = new array[n+1];
    while (i > 0){
        tags[i] = t;
        t = trellis[i][t].backpointer;
        i--;
    }
    return tags;
}
```

Today’s key concepts

HMM taggers
Learning HMMs from labeled text
Viterbi for HMMs
Dynamic programming
Independence assumptions in HMMs
The trellis

Trigram HMMs

In a Trigram HMM, transition probabilities are of the form:

\[
P(t^{(i)} = t_j | t^{(i-1)} = t_j, t^{(i-2)} = t_k)\]

The i-th tag in the sequence influences the probabilities of the (i+1)-th tag and the (i+2)-th tag:

\[
\ldots P(t^{(i+1)} | t^{(i)}, t^{(i-1)}) \ldots P(t^{(i+2)} | t^{(i+1)}, t^{(i)})
\]

Hence, each row in the trellis for a trigram HMM has to correspond to a pair of tags — the current and the preceding tag:

(assuming notation)

trellis[i][j,k]: word w^{(i)} has tag t_j, word w^{(i-1)} has tag t_k

The trellis now has T^2 rows.

But we still need to consider only T transitions into each cell, since the current word’s tag is the next word’s preceding tag:

Transitions are only possible from trellis[i][j,k] to trellis[i+1][l,j]