Lecture 11: Viterbi and Forward Algorithms

Kai-Wei Chang
CS @ University of Virginia
kw@kwchang.net

Course webpage: http://kwchang.net/teaching/NLP16
Quiz 1

- Max: 24; Mean: 18.1; Median: 18; SD: 3.36
This lecture

- Two important algorithms for inference
  - Forward algorithm
  - Viterbi algorithm
Three basic problems for HMMs

- **Likelihood** of the input:
  - Forward algorithm
    - How likely the sentence “I love cat” occurs

- **Decoding** (tagging) the input:
  - Viterbi algorithm
    - POS tags of “I love cat” occurs

- **Estimation** (learning):
  - Find the best model parameters
    - Case 1: supervised – tags are annotated
      - Maximum likelihood estimation (MLE)
    - Case 2: unsupervised -- only unannotated text
      - Forward-backward algorithm

How to learn the model?
Likelihood of the input

- How likely a sentence “I love cat” occur
- Compute $P(w \mid \lambda)$ for the input $w$ and HMM $\lambda$

- Remember, we model $P(t, w \mid \lambda)$
- $P(w \mid \lambda) = \sum_t P(t, w \mid \lambda)$

**Marginal probability:**
Sum over all possible tag sequences
Likelihood of the input

- How likely a sentence “I love cat” occur
  \[ P(w | \lambda) = \sum_t P(t, w | \lambda) = \sum_t \prod_{i=1}^{n} P(w_i | t_i)P(t_i | t_{i-1}) \]
- Assume we have 2 tags N, V
  \[ P(“I love cat” | \lambda) = P(“I love cat”,”NNN” | \lambda) + P(“I love cat”,”NNV” | \lambda) + P(“I love cat”,”NVN” | \lambda) + P(“I love cat”,”VNN” | \lambda) + P(“I love cat”,”VVN” | \lambda) + P(“I love cat”,”VVV” | \lambda) \]
- Now, let’s write down \[ P(“I love cat” | \lambda) \] with 45 tags…
Goal: \( P(w | \lambda) = \sum_t \prod_{i=1}^{n} P(w_i | t_i) P(t_i | t_{i-1}) \)

\( P(t_2 = 2 | t_1 = 1) \)

\( P(t_3 = 1 | t_2 = 1) \)

\( P(w_3 | t_3 = 1) \)

\( \lambda \) is the parameter set of HMM. Let’s ignore it in some slides for simplicity’s sake
Trellis diagram

- $P(\text{“I eat a fish”}, \text{NVVA })$
Trellis diagram

\[ \sum_t \prod_{i=1}^{n} P(w_i | t_i) P(t_i | t_{i-1}) : \text{sum over all paths} \]
Dynamic programming

- Recursively decompose a problem into smaller sub-problems
- Similar to mathematical induction
  - **Base step**: initial values for $i = 1$
  - **Inductive step**: assume we know the values for $i = k$, let’s compute $i = k + 1$
Inductive step: from $i = k$ to $i = k + 1$

- $t^k$: tag sequence with length $k$, $w^k = w_1, w_2 \ldots w_k$
- $\sum t^k P(t^k, w^k) = \sum_q \sum t^{k-1} P(t^{k-1}, w^k, t_k = q)$
Forward algorithm

- **Inductive step:** from $i = k$ to $i = k+1$

\[ \sum_{t_k} P(t^k, w) = \sum_q P(w^k, t_k = q) \]

\[ P(w^k, t_k = q) = \sum_{q'} P(w^k, t_{k-1} = q', t_k = q) \]

\[ = \sum_{q'} P(w^{k-1}, t_{k-1} = q')P(t_k = q \mid t_{k-1} = q')P(w_k \mid t_k = q) \]
Forward algorithm

- **Inductive step**: from \(i = k\) to \(i = k+1\)

\[
\sum_{t_k} P(t^k, w) = \sum_q P(w^k, t_k = q)
\]

\[
P(w^k, t_k = q) = \sum_{q'} P(w^k, t_{k-1} = q', t_k = q)
\]

\[
= \sum_{q'} P(w^{k-1}, t_{k-1} = q') P(t_k = q \mid t_{k-1} = q') P(w_k \mid t_k = q)
\]

Let’s call it \(\alpha_k(q)\)

This is \(\alpha_{k-1}(q')\)
Forward algorithm

- **Inductive step:** from $i = k$ to $i = k + 1$

- $\alpha_k(q) = \sum_{q'} \alpha_{k-1}(q') P(t_k = q \mid t_{k-1} = q') P(w_k \mid t_k = q)$
Forward algorithm

- **Inductive step:** from $i = k$ to $i = k + 1$

  $$\alpha_k(q) = \sum_{q', \alpha_{k-1}(q')} P(t_k = q \mid t_{k-1} = q') P(w_k \mid t_k = q)$$
  $$\quad = P(w_k \mid t_k = q) \sum_{q', \alpha_{k-1}(q')} P(t_k = q \mid t_{k-1} = q')$$
Forward algorithm

- **Base step:** $i=0$
- $\alpha_1(q) = P(w_1 \mid t_1 = q)P(t_1 = q \mid t_0)$

Initial probability $p(t_1 = q)$
Implementation using an array

Use an $n \times T$ table to keep $\alpha_k(q)$

From Julia Hockenmaier, Intro to NLP
Implementation using an array

Initial:

\[
\text{Trellis}[1][q] = P(w_1 \mid t_1 = q)P(t_1 = q \mid t_0)
\]
Implementation using an array

\[
\alpha_k(q) = P(w_k \mid t_k = q) \sum_{q'} \alpha_{k-1}(q') P(t_k = q \mid t_{k-1} = q')
\]

Induction:

\[
\alpha_k(q) = P(w_k \mid t_k = q) \sum_{q'} \alpha_{k-1}(q') P(t_k = q \mid t_{k-1} = q')
\]
The forward algorithm (Pseudo Code)

```python
forward( w_1...n)
    for t (1...T){
        trellis[1][t].fwd = p_init[t] × p_emit[t][w_1]
    }
    for i (2...n){
        for t (1....T){
            trellis[i][t].fwd = 0
            for t’ (1...T)
                trellis[i][t].fwd += trellis[i-1][t’].fwd × p_trans[t’][t]
                trellis[i][t].fwd ×= p_emit[t][w_i]
        }
    }
    p_w = 0
    for t (1...T)
        p_w += trellis[n][t].fwd
    return p_w
```
Jason’s ice cream

|       | p(…|C) | p(…|H) | p(…|START) |
|-------|-------|-------|------------|
| (1|…)  | 0.5   | 0.1   |            |
| (2|…)  | 0.4   | 0.2   |            |
| (3|…)  | 0.1   | 0.7   |            |
| (C|…)  | 0.8   | 0.2   | 0.5        |
| (H|…)  | 0.2   | 0.8   | 0.5        |

P(”1,2,1”)?

Scroll to the bottom to see a graph of what states and transitions the model thinks are likely on each day. Those likely states and transitions can be used to reestimate the red probabilities (this is the “forward-backward” or Baum-Welch algorithm), incr...
Three basic problems for HMMs

- **Likelihood** of the input:
  - Forward algorithm

- **Decoding** (tagging) the input:
  - Viterbi algorithm

- **Estimation** (learning):
  - Find the best model parameters
    - Case 1: supervised – tags are annotated
      - Maximum likelihood estimation (MLE)
    - Case 2: unsupervised -- only unannotated text
      - Forward-backward algorithm

How likely the sentence ”I love cat” occurs

POS tags of ”I love cat” occurs

How to learn the model?
Prediction in generative model

- **Inference:** What is the most likely sequence of tags for the given sequence of words $w$?

- What are the latent states that most likely generate the sequence of word $w$?
Tagging the input

- Find best tag sequence of “I love cat”
- Remember, we model $P(t,w | \lambda)$
- $t^* = \arg \max_t P(t, w | \lambda)$

Find the best one from all possible tag sequences
Tagging the input

- Assume we have 2 tags N, V
- Which one is the best?

\[ P(\text{"I love cat"}, "NNN" \mid \lambda), P(\text{"I love cat"}, "NNV" \mid \lambda), \]
\[ P(\text{"I love cat"}, "NNV" \mid \lambda), P(\text{"I love cat"}, "NVV" \mid \lambda), \]
\[ P(\text{"I love cat"}, "VNN" \mid \lambda), P(\text{"I love cat"}, "VNV" \mid \lambda), \]
\[ P(\text{"I love cat"}, "VVN" \mid \lambda), P(\text{"I love cat"}, "VVV" \mid \lambda) \]

- Again! We need an efficient algorithm
Trellis diagram

- **Goal:** $\arg \max_t \prod_{i=1}^n P(w_i | t_i) P(t_i | t_{i-1})$

- $P(t_2 = 2 | t_1 = 1)$
- $P(t_3 = 1 | t_2 = 1)$
- $P(w_3 | t_3 = 1)$

$i = 1$  $i = 2$  $i = 3$  $i = 4$
Trellis diagram

- **Goal:** $\arg \max \prod_{i=1}^{n} P(w_i|t_i)P(t_i | t_{i-1})$
- **Find the best path!**
Dynamic programming again!

- Recursively decompose a problem into smaller sub-problems
- Similar to mathematical induction
  - **Base step**: initial values for $i = 1$
  - **Inductive step**: assume we know the values for $i = k$, let’s compute $i = k + 1$
Viterbi algorithm

- **Inductive step:** from $i = k$ to $i = k+1$

- $t^k$: tag sequence with length $k$, $w^k = w_1, w_2 \ldots w_k$

- $\max_{t^k} P(t^k, w^k) = \max_q \max_{t^{k-1}} P(t^{k-1}, t_k = q, w^k)$
Viterbi algorithm

- **Inductive step**: from \( i = k \) to \( i = k + 1 \)

\[
\max_{t_k} \max_{t_{k-1}} P(t_{k-1}, t_k = q, w^k)
\]

\[
= \max_{q'} \max_{t_{k-1}} P(t_{k-2}, t_k = q, t_{k-1} = q', w^k)
\]

\[
= \max_{q'} \max_{t_{k-1}} P(t_{k-2}, t_{k-1} = q', w^{k-1}) P(t_k = q, t_{k-1} = q') P(w_k | t_k = q)
\]

Let's call it \( \delta_k(q) \)

This is \( \delta_{k-1}(q') \)
Viterbi algorithm

- **Inductive step:** from $i = k$ to $i = k+1$

\[
\delta_k(q) = \max_{q'} \delta_{k-1}(q') P(t_k = q \mid t_{k-1} = q') P(w_k \mid t_k = q)
\]
Viterbi algorithm

- **Inductive step:** from \( i = k \) to \( i = k + 1 \)

\[
\delta_k(q) = \max_{q'} \delta_{k-1}(q') P(t_k = q \mid t_{k-1} = q') P(w_k \mid t_k = q)
\]
\[
= P(w_k \mid t_k = q) \max_{q'} \delta_{k-1}(q') P(t_k = q \mid t_{k-1} = q')
\]
Viterbi algorithm

- **Base step**: $i=0$

- $\delta_1(q) = P(w_1 \mid t_1 = q)P(t_1 = q \mid t_0)$

Initial probability $p(t_1 = q)$
Implementation using an array

Initial:
Trellis[1][q] = \( P(w_1 \mid t_1 = q)P(t_1 = q \mid t_0) \)
Implementation using an array

Induction:

\[ \delta_k(q) = P(w_k \mid t_k = q) \max_{q'} \delta_{k-1}(q') P(t_k = q \mid t_{k-1} = q') \]
Retrieving the best sequence

- Keep one **backpointer**

\[
\text{trellis}[n][i] = \max(trellis[n-1][j]P(t_i | t_j))
\]
The Viterbi algorithm (Pseudo Code)

Viterbi( w_1...n)
   for t (1...T) // INITIALIZATION
      trellis[1][t].viterbi = p_init[t] \times p_emit[t][w_1]
   for i (2...n) // RECURSION
      for t (1...T){
         trellis[i][t] = 0
         for t’ (1...T){
            tmp = trellis[i-1][t’].viterbi \times p_trans[t’][t]
            if (tmp > trellis[i][t].viterbi){
               trellis[i][t].viterbi = tmp
               trellis[i][t].backpointer = t’
            } 
            trellis[i][t].viterbi \times= p_emit[t][w_i]}
      }
   t_max = NULL, vit_max = 0; // FINISH
   for t (1...T)
      if (trellis[n][t].vit > vit_max)\{t_max = t; vit_max = trellis[n][t].value \}
   return unpack(n, t_max);

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

Max instead of sum
forward(w_{1..n})
    for t (1...T) {
        trellis[1][t].fwd = p_init[t] \times p_emit[t][w_1]
    }
    for i (2...n) {
        for t (1...T) {
            trellis[i][t].fwd = 0
            for t’ (1...T)
                trellis[i][t].fwd += trellis[i-1][t’].fwd \times p_trans[t’][t]
                trellis[i][t].fwd \times p_emit[t][w_i]
        }
    }
    p_w = 0
    for t (1...T)
        p_w += trellis[n][t].fwd
    return p_w

Viterbi(w_{1..n})
    for t (1...T) // INITIALIZATION
        trellis[1][t].viterbi = p_init[t] \times p_emit[t][w_1]
    for i (2...n) // RECURSION
        for t (1...T) {
            trellis[i][t] = 0
            for t’ (1...T) {
                tmp = trellis[i-1][t’].viterbi \times p_trans[t’][t]
                if (tmp > trellis[i][t].viterbi) {
                    trellis[i][t].viterbi = tmp
                    trellis[i][t].backpointer = t’
                }
                trellis[i][t].viterbi \times p_emit[t][w_i]}
            t_max = NULL, vit_max = 0; // FINISH
            for t (1...T)
                if (trellis[n][t].vit > vit_max) {t_max = t; vit_max = trellis[n][t].value }
            return unpack(n, t_max);
Jason’s ice cream

Best tag sequence for P(”1,2,1”)?

|     | p(…|C) | p(…|H) | p(…|START) |
|-----|------|-------|------------|
| (1|…)  | 0.5  | 0.1    |            |
| (2|…)  | 0.4  | 0.2    |            |
| (3|…)  | 0.1  | 0.7    |            |
| (C|…)  | 0.8  | 0.2    | 0.5        |
| (H|…)  | 0.2  | 0.8    | 0.5        |

#cones

Scroll to the bottom to see a graph of what states and transitions the model thinks are likely on each day. Those likely states and transitions can be used to reestimate the red probabilities (this is the “forward-backward” or Baum-Welch algorithm), incr
Trick: computing everything in log space

- Homework:
  - Write **forward** and **Viterbi** algorithm in log-space

- Hint: you need a function to compute \( \log(a+b) \)
Three basic problems for HMMs

- **Likelihood** of the input:
  - Forward algorithm
  - How likely the sentence ”I love cat” occurs

- **Decoding** (tagging) the input:
  - Viterbi algorithm
  - POS tags of ”I love cat” occurs

- **Estimation** (learning):
  - Find the best model parameters
  - Case 1: supervised – tags are annotated
    - Maximum likelihood estimation (MLE)
  - Case 2: unsupervised -- only unannotated text
    - Forward-backward algorithm
  - How to learn the model?