Natural Language Processing
Unit 2 – Tagging Problems and HMM

Anantharaman Narayana Iyer

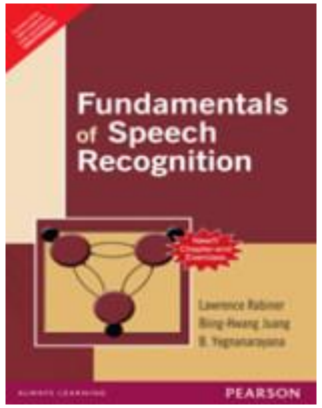
narayana dot Anantharaman at gmail dot com

5th Sep 2014

References

- A tutorial on hidden Markov models and selected applications in speech recognition, L Rabiner (cited by over 19395 papers!)
- Fundamentals of speech recognition by L Rabiner, et al, Pearson
- Discrete-Time processing of Speech Signals John R Deller et al, IEEE Press Classic release
- Week 2 Language Modeling Problem, Parameter Estimation in Language Models lectures of Coursera course on Natural Language Processing by Michael Collins
- Youtube videos: https://www.youtube.com/watch?v=TPRoLreU9lA

Hidden Markov Model (HMM)
0:04 / 14:29
(ML 14.4) Hidden Markov models (HMMs) (part 1)
mathematicalmonk's channel
Subscribe
18,101
57,819



Objective of today’s and the next class

- Introduce, using simple examples, the theory of HMM
- Mention the applications of HMM
- Forward, Backward algorithm, Viterbi algorithm
- Introduce the tagging problem and describe its relevance to NLP
- Describe the solutions to the tagging problems using HMM
- Pre requisite for this class:
    - Mathematics for Natural Language Processing (Covered in Unit 1)
    - Markov processes
    - HMM intuition (Covered yesterday)

Motivation: Extending Markov Processes

- Suppose we are using Markov Processes to perform spelling error detection/correction, given the input as a typed word
  - In this example, the input alphabet is an unambiguous symbol that can be mapped to the underlying alphabet directly
- What happens when the input is not a typed word but is a handwritten word?
  - In this case, the input is a sequence of pen strokes, curves, angles, lines etc. The input itself is stochastic in nature and hence mapping this to an underlying set of symbols is not straightforward
- Hence for such inputs, Markov processes fall short as they make **two simplifying assumptions** that don't hold any longer:
  - Markov model assumes that the input is a sequence of symbols
  - Input symbols directly correspond to the states of the system
- These assumptions are too restrictive to many real world problems and hence the model needs to be extended

Hidden Markov Models

- Hidden Markov Models provide a more powerful framework as they allow the states to be separated from the input without requiring a direct mapping.
- In HMM, the states are **hidden and abstract** while the inputs are **observables**
    - Eg, in handwriting recognition, the input that corresponds to “A” can be written in many ways, while the representation of this in the computer is unique and deterministic.
- Thus, the input space may be discrete or continuous variables that are observed while the states are discrete, abstract and hidden
- The observation is a probabilistic function of the state. The resulting model is a doubly embedded stochastic process.
- The underlying stochastic process is not directly observable but can be observed only through another set of stochastic processes that produce the sequence of observations

Hidden Markov Model: Formalization

- HMM is a stochastic finite automaton specified by a 5-tuple:

  HMM = **(N, M, A, B, π)** where:

  N = Number of states (hidden). In general all states are fully connected (Ergodic Model). However, other topologies are also found useful, for example in speech applications

  M = Number of distinct observation symbols per state, where individual symbols are from the set: V = {v1, v2, …v_k}

  A = State transition matrix where the elements a_{ij} correspond to the probability of transitioning to state j from state i. That is: $P(q_{i+1} = j \mid q_i = i)$, $1 \le i, j \le N$

  B = Emission probability distribution of the form $\{b_i(k)\}$ where $b_i(k)$ refers to the probability of emitting the observable k when the HMM is in state i . This is of the form: $b_i(k) = P(x_t = v_k \mid q_i = i)$, $1 \le i \le N$, $1 \le k \le M$

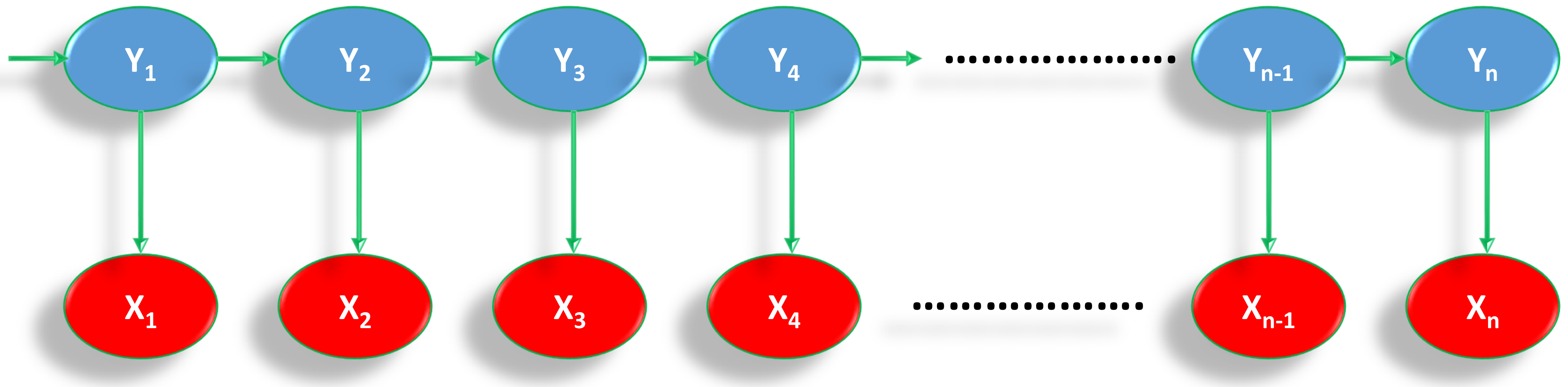
  π = probability of initial state distribution of the form $\{\pi_i\}$ where $\pi_i = P(q_1 = i)$

- The above can be represented by a compact notation:

$$\lambda = (A, B, \pi)$$

Trellis Diagram

- An HMM can be graphically depicted by Trellis diagram



Another example: Coin tossing

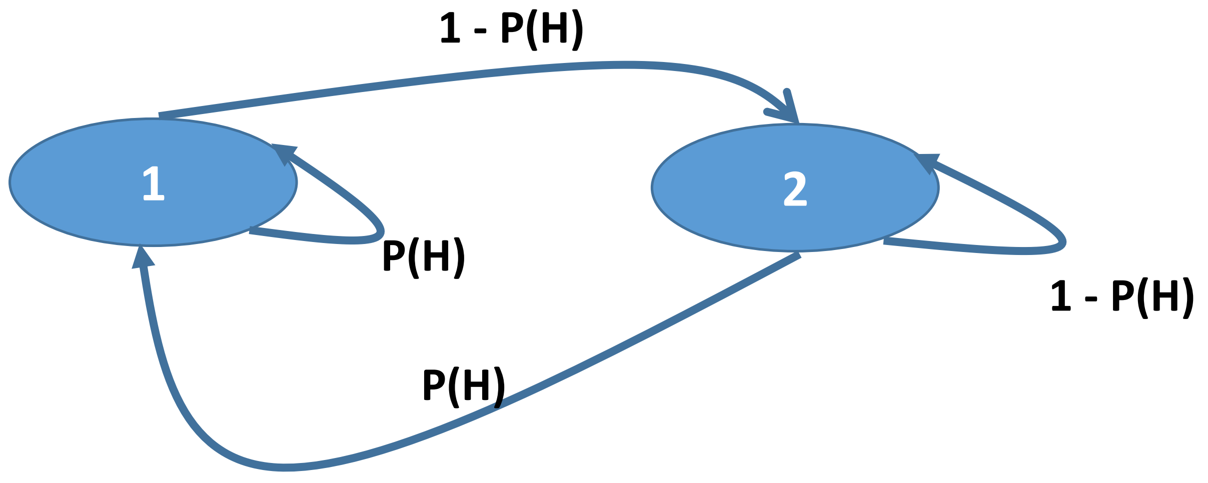


Fig (a)

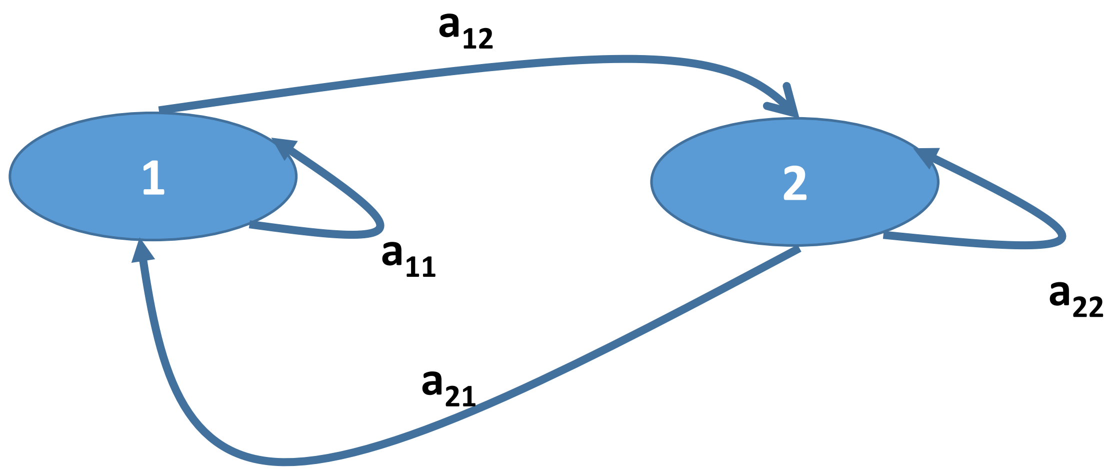


Fig (b)

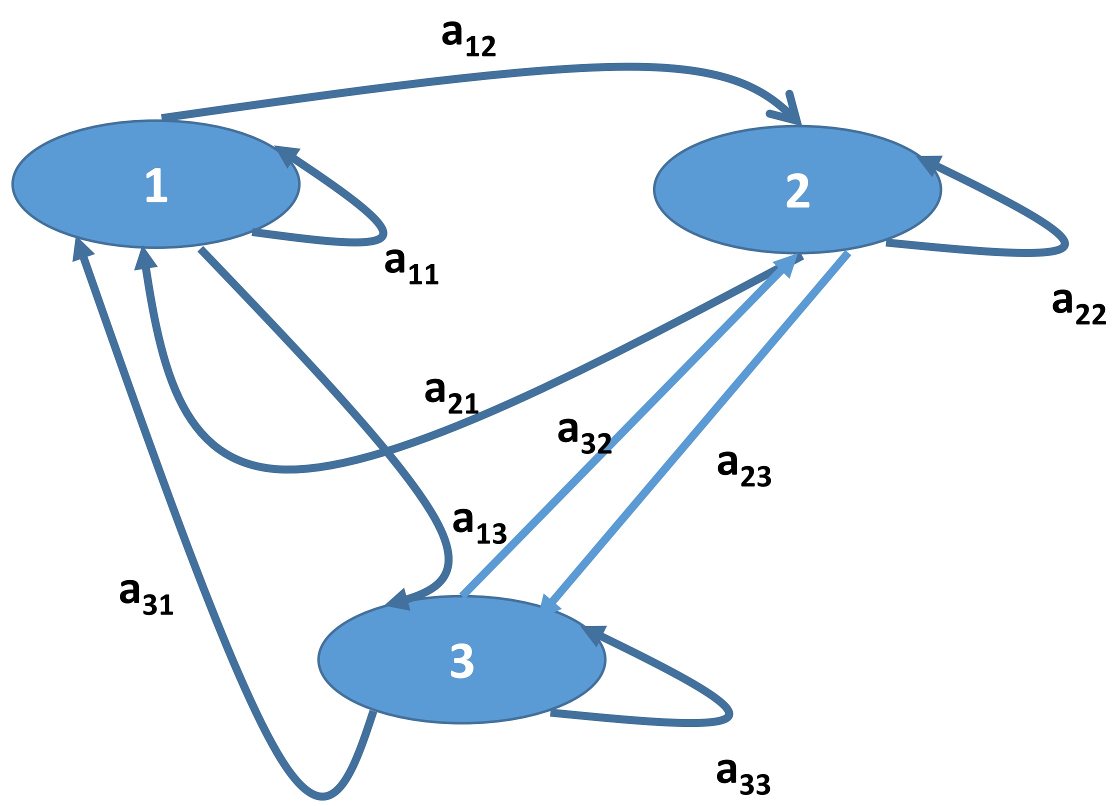


Fig (c)

	State 1	State 2	State 3
P(H)	0.5	0.75	0.25
P(T)	0.5	0.25	0.75

Exercise

- Consider the 3 state coin tossing model as shown in the previous slide and suppose the state transition probabilities have a uniform distribution of 1/3 for each a_{ij}.

  Suppose you observe a sequence: O = (H H H H T H T T T T)

  What state sequence is most likely? What is the probability of the observation sequence and this most likely state sequence?

- Solution:

  As all state transitions are equally probable, the most likely sequence is the one where the probability of each individual observation is maximum. For H, the most likely state is 2 and for T it is 3. Hence the state sequence that is most likely is: q = (2 2 2 2 3 2 3 3 3 3)

  $P(O, q \mid \lambda) = (0.75)^{10} (1/3)^{10}$

HMM as a Generator

- One can view the HMM as a generator that emits the observations based on the emission probability distribution from a given state and state transitions occur in a sequence as per the transition probability
- In reality HMM doesn’t “generate” the observable.
- But we imagine the HMM to be generating the observation sequence and the state sequence that has the maximum probability of generating the observed sequence is considered the output of HMM for those problems where we are required to determine the hidden state sequence

Representing the model $\lambda = (A, B, \pi)$ – JSON example

```
{
  "hmm":
       {
         "A": {
           "Coin 1" : {"Coin 1": 0.1, "Coin 2": 0.3, "Coin 3": 0.6},
           "Coin 2" : {"Coin 1": 0.8, "Coin 2": 0.13, "Coin 3": 0.07},
           "Coin 3" : {"Coin 1": 0.33, "Coin 2": 0.33, "Coin 3": 0.34}
        },
         "B": {
             "Coin 1" : {"Heads": 0.15, "Tails": 0.85},
             "Coin 2" : {"Heads": 0.5, "Tails": 0.5},
             "Coin 3" : {"Heads": 0.5, "Tails": 0.5}
         },
         "pi": {"Coin 1": 0.33, "Coin 2": 0.33, "Coin 3": 0.34}
       }
}
```

Applying HMM – the three basic problems

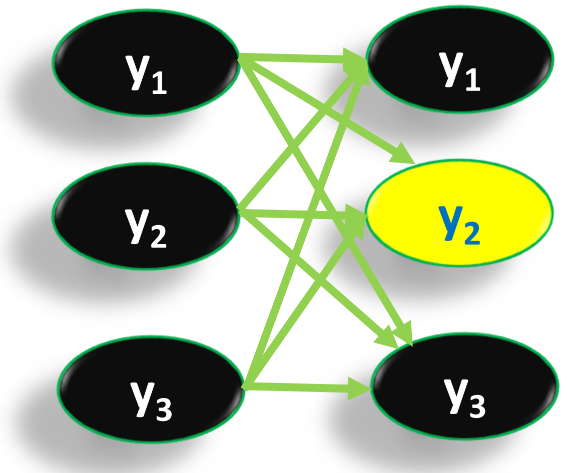
1. Given the observation sequence X and the model λ = (A, B, π), how do we efficiently compute P(X| λ), the probability of an observation sequence given the model?
    - Evaluation Problem
2. Given the observation sequence X and the model λ = (A, B, π), how do we determine the state sequence Y that best explains the observation?
    - Decoding Problem
3. How do we adjust the model parameters λ = (A, B, π)to maximize P(X| λ), the probability of an observation sequence given the model?
    - Training Problem

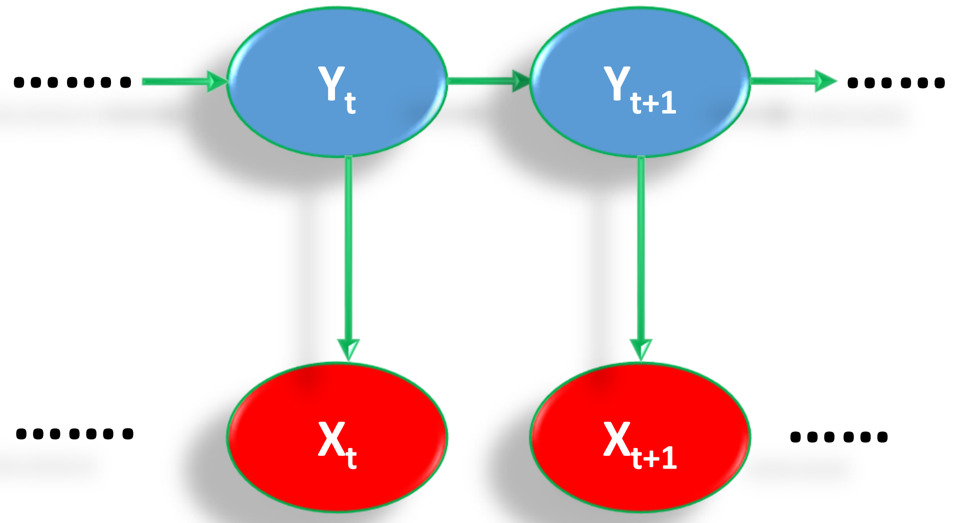
Dynamic Programming Characteristics

- Dynamic Programming is an algorithm design technique that can improve the performance significantly compared to brute force methods
  - Define the problem in a way that the solution can be obtained by solving sub-problems and reusing the partial solutions obtained while solving the sub problems.
- Overlapping sub problems: A recursive solution to the main problem contains many repeating sub problems
- Optimal sub structure: Solutions to the sub problems can be combined to provide an optimal solution to the larger problem

Forward Algorithm - Intuition

- Our goal is to determine the probability of a sequence of observations ($X_1, X_2, \ldots, X_n$) given λ
- In the forward algorithm approach, we divide the sequence X in to sub-sequences, compute the probabilities, store them in the table for later use.
- The probability of a larger sequence is obtained by combining the probabilities of these smaller sequences.
- Specifically, we compute the joint probability of a sub-sequence starting from time t = 1 where the sub-sequence ends on a state y. We compute: $P(X_{1:t}, Y_t \mid \lambda)$
- We then compute $P(X_{1:n} \mid \lambda)$ by marginalizing Y





Forward Algorithm

- Goal: Compute P(Y_k, $X_{1:k}$) assuming the model parameters to be known
- Approach: As we know the emission and transition probabilities, we try to factorize the joint distribution P(Y_k, $X_{1:k}$) in terms of the known parameters and solve. In order to implement efficiently we use dynamic programming where a large problem is solved by solving the overlapping sub-problems and combining the solution. To do this set up the recursion.

We can write: $X_{1:k} = X_{1,} X_2 \dots X_{k-1}, X_k$

From sum rule we know: P(X = x_i) = $\sum_j P(X = x_i, Y = y_j)$

$$P(Y_k, X_{1:k}) = \sum_{y_{k\_1}}^{m} P\,(Y_k, Y_{k-1}, X_{1:k})$$

$$P(Y_k, X_{1:k}) = \sum_{y_{k\_1}}^{m} P\,(X_{1:k-1}, Y_{k-1}, Y_k, X_k)$$

From product rule the above factorizes to:

$$\sum_{y_{k\_1}}^{m} P\,(X_{1:k-1}) P(Y_{k-1}|X_{1:k-1}) P(Y_k|Y_{k-1}, X_{1:k-1})\, P(X_k\,|Y_k, Y_{k-1}, X_{1:k-1})$$

= $\sum_{y_{k\_1}}^{m} P\,(X_{1:k-1}) P(Y_{k-1}|X_{1:k-1}) P(Y_k|Y_{k-1})\, P(X_k|Y_k)$

We can write: $\alpha_k(Y_k) = \sum_{y_{k\_1}}^{m} P\,(Y_k|Y_{k-1})\, P(X_k|Y_k) \alpha_{k-1}(Y_{k-1})$

Initialization: $\alpha_1(Y_1) = P(Y_1, X_1) = P(Y_1)$ P(X_1|Y_1)

We can now compute the different α values

Forward Algorithm Summary

Multiplications

1. $\alpha_1(i) = \pi_i b_i(O_1),\ 1 \le i \le N$

$= N$

2. for $t = 1, 2, \ldots, T\text{-}1,\ 1 \le j \le N$

$\alpha_{t+1}(j) = [\Sigma_{i=1 \text{ to } N}\ \alpha_t(i) * a_{ij}] * b_j(O_{t+1})$

$= (N+1)N(T\text{-}1)$

3. Finally we have:

$P(O|\ \lambda) = \Sigma_{i=1 \text{ to } N}\ \alpha_T(i)$

$= 0$

Total:
$N+(N+1)N(T\text{-}1)$

Forward Algorithm: Implementation

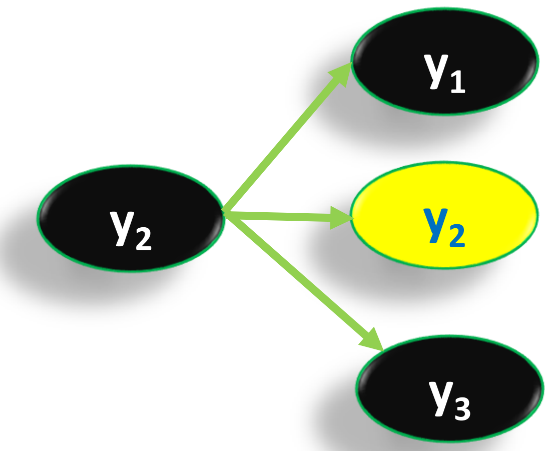
```
def forward(self, obs):
    self.fwd = [{}]
    for y in self.states:
        self.fwd[0][y] = self.pi[y] * self.B[y][obs[0]] # Initialize base cases
    for t in range(1, len(obs)):
        self.fwd.append({})
        for y in self.states:
            self.fwd[t][y] = sum((self.fwd[t-1][y0] * self.A[y0][y] * self.B[y][obs[t]]) for y0 in
self.states)
    prob = sum((self.fwd[len(obs) - 1][s]) for s in self.states)
    return prob
```

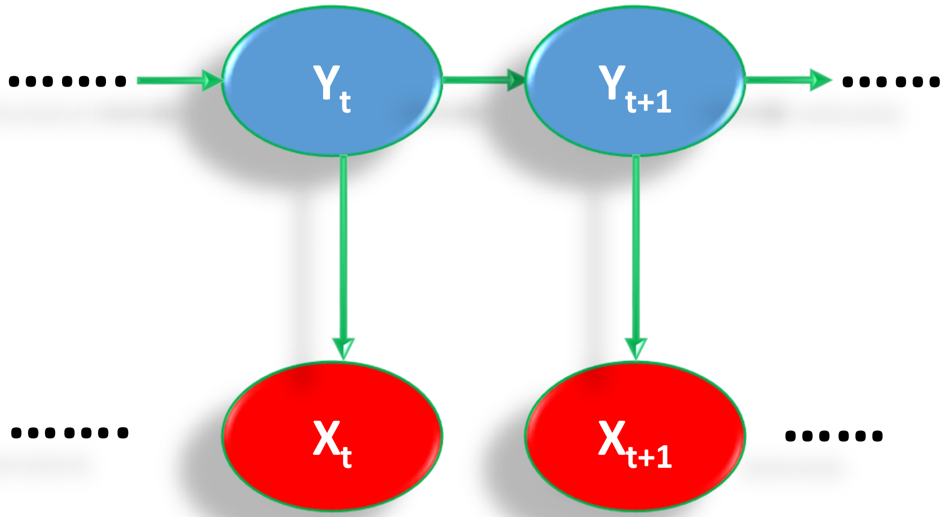
Demo – Let’s toss some coins!

- And see how probable getting some 10 consecutive heads is!

Backward Algorithm - Intuition

- Our goal is to determine the probability of a sequence of observations ($X_{k+1,}$ X_{k+2}, …, X_n|Y_k, λ)
- Given that the HMM has seen k observations and ended up in a state Y_k = y, compute the probability of the remaining part: $X_{k+1,}$ X_{k+2}, …, X_n
- We form the sub-sequences starting from the last observation X_n and proceed backward to the first.
- Specifically, we compute the conditional probability of a sub-sequence starting from k+1 and ending in n, where the state at k is given.
- We can compute P($X_{1:n}$ | λ) by marginalizing Y. The probability of an observation sequence computed by backward algorithm will be equal to that computed with forward algorithm.





Backward Algorithm - Derivation

- Goal: Compute P($X_{k+1:n}$|Y_k) assuming the model parameters to be known
- P(Xk$_{+1:n}$|Yk) $= \sum_{y_{k_{+}1}}^{m} P\,(X_{k\ +\ 1_{:}\ n}, Y_{k+1}|Yk)$
- To be explained on the board as typing a lot of equations is a pain!
- Basically we need to do the following:
  - Express P(Xk$_{+1:n}$|Yk) in a form where we can combine it with another random variable and marginalize to get the required probability. We do this because our strategy is to find a recurrence.
  - Factorize the joint distribution obtained as above using chain rule
  - Apply Markov assumptions and conditional independence property in order to bring the factors to a form where the factors are known from our model
  - Determine the base cases and termination conditions

Backward Algorithm: Summary

<u>Multiplications</u>

1. $\beta_T(i) = 1,\ 1 \leq i \leq N$

   = 0

2. for $t = T\text{-}1, T\text{-}2, \ldots, 1,\ 1 \leq i \leq N$

   $\beta_t(i) = \Sigma_{j=1 \text{ to } N}\ a_{ij} * b_j(O_{t+1}) * \beta_{t+1}(j)$

   = (2N)N(T-1)

3. Finally we have:

   $P(O|\lambda) = \Sigma_{i=1 \text{ to } N}\ \pi_i * b_i(O_1) * \beta_1(i)$

   = 2N

Total:
2N+(2N)N(T-1)

Backward Algorithm Implementation

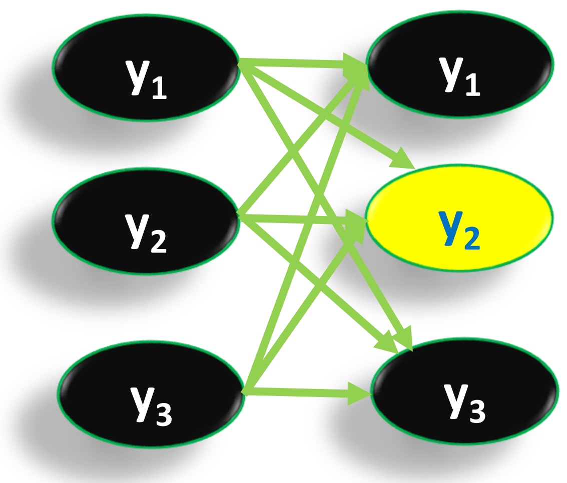
```
def backward(self, obs):
    self.bwk = [{} for t in range(len(obs))]
    T = len(obs)
    for y in self.states:
        self.bwk[T-1][y] = 1
    for t in reversed(range(T-1)):
        for y in self.states:
            self.bwk[t][y] = sum((self.bwk[t+1][y1] * self.A[y][y1] *
self.B[y1][obs[t+1]]) for y1 in self.states)
    prob = sum((self.pi[y]* self.B[y][obs[0]] * self.bwk[0][y]) for y in self.states)
    return prob
```

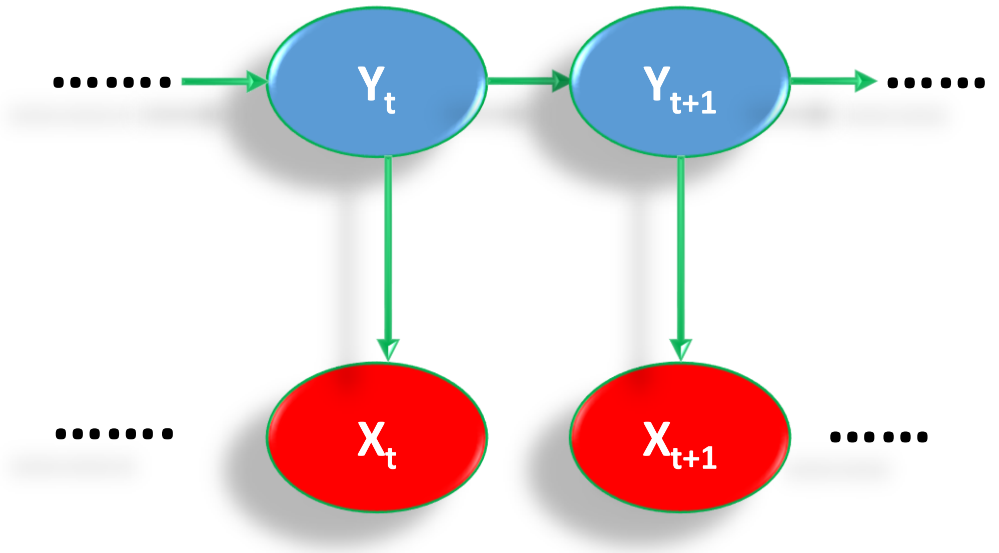
Demo – Let’s run the backward algorithm!

- And see if it computes the same value of probability for the given observation as computed by the forward algorithm!

Viterbi Algorithm - Intuition

- Our goal is to determine the most probable state sequence for a given sequence of observations ($X_{1,}$ X_2, …, X_n) given λ
- This is a decoding process where we discover the hidden state sequence looking at the observations
- Specifically, we need: $\text{argmax}_Y\ P(X_{1:t} | Y_{1:t}, \lambda)$. This is equivalent to finding $\text{argmax}_Y\ P(X_{1:t}, Y_{1:t} \mid \lambda)$
- In the forward algorithm approach, we computed the probabilities along each path that led to the given state and summed the probabilities to get the probability of reaching that state regardless of the path taken.
- In Viterbi we are interested in only a specific path that maximizes the probability of reaching the required state. Each state along this path (the one that yields max probability) forms the sequence of hidden states that we are interested in.





Viterbi Algorithm

- Finding the best single sequence means computing $\text{argmax}_Q P(Q|O,\lambda)$, equivalent to $\text{argmax}_Q P(Q,O|\lambda)$
- The Viterbi algorithm (dynamic programming) defines $\delta_j(t)$, i.e., the highest probability of a single path of length t which accounts for the observations and ends in state S_j

$$\delta_j(t) = \max_{q_1,q_2,\dots,q_{t-1}} P(q_1 q_2 \dots q_t = j, O_1 O_2 \dots O_t|\lambda)$$

- By induction

$$\begin{aligned} \delta_j(1) &= \pi_j b_{jO_1} & 1 \le j \le N \\ \delta_j(t+1) &= \left(\max_i \delta_i(t) a_{ij}\right) b_{jO_{t+1}} & 1 \le t \le T-1 \end{aligned}$$

- With backtracking (keeping the maximizing argument for each t and j) we find the optimal solution

Viterbi Implementation

```
def viterbi(self, obs):
    vit = [{}]
    path = {}
    for y in self.states:
        vit[0][y] = self.pi[y] * self.B[y][obs[0]]
        path[y] = [y]
    for t in range(1, len(obs)):
        vit.append({})
        newpath = {}
        for y in self.states:
            (prob, state) = max((vit[t-1][y0] * self.A[y0][y] * self.B[y][obs[t]], y0) for y0 in self.states)
            vit[t][y] = prob
            newpath[y] = path[state] + [y]
            path = newpath
    (prob, state) = max((vit[len(obs) - 1][y], y) for y in self.states)
    return (prob, path[state])
```

Demo – You can’t hide from my algorithm!

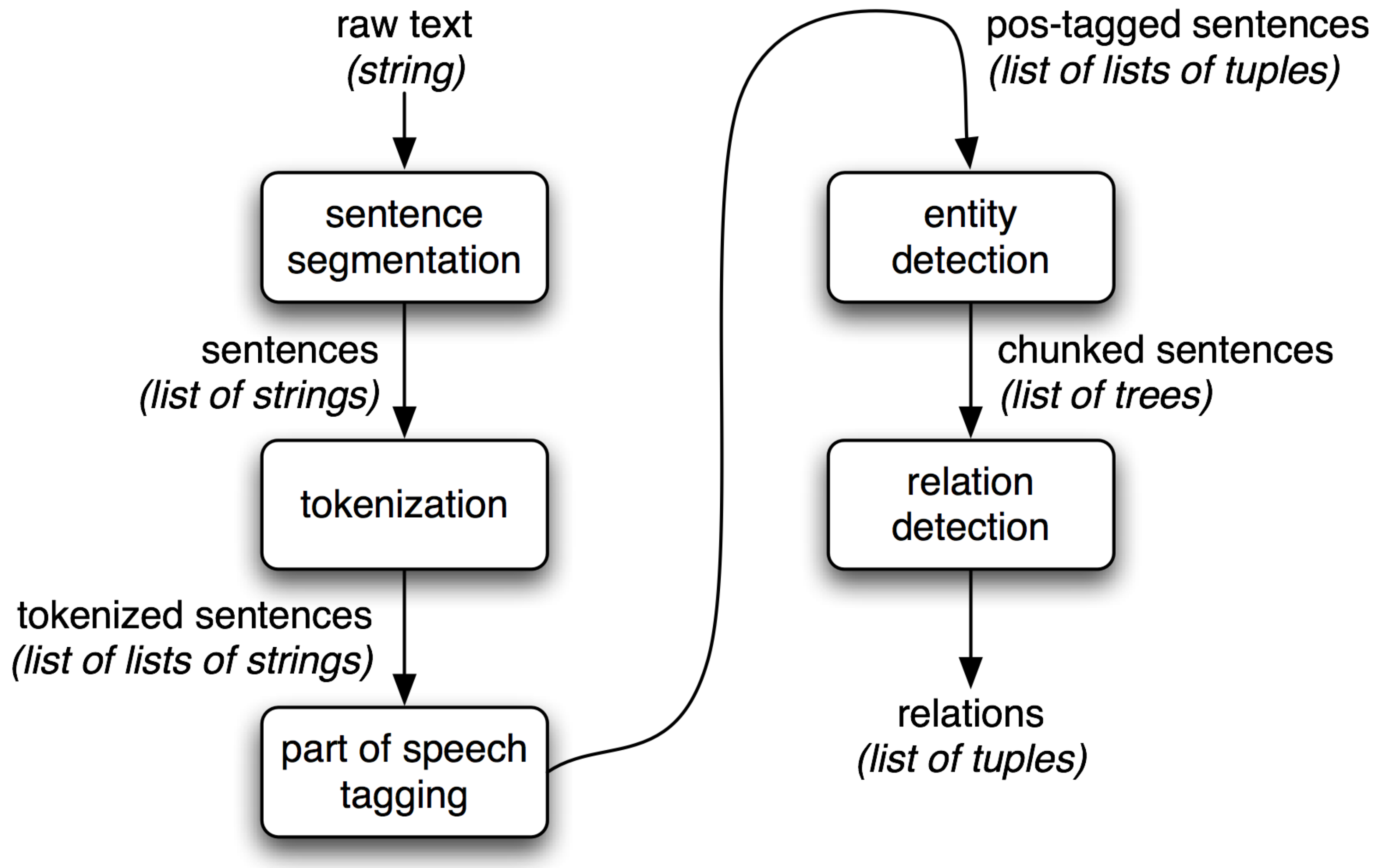
- If we are to guess this would we have concluded on the same sequence?

Forward Backward Algorithm

Tagging Problems

- Purpose of the tagging is to facilitate further analysis
  - Information extraction – LinkedIn resume extraction example
- Tags are the metadata that allows the given text to be processed further
- E.g. POS tagging annotates each word in a text with a part of speech tag. This helps finding noun/verb phrases and can be used for parsing
- E.g NER

Applications of taggers: POS, NER



Examples of Tagging Problem

- Microsoft will seek to draw more people to its internet-based services with two new mid-range smartphones it unveiled, including one designed to help people take better selfies.
- [('microsoft', 'NN'), ('will', 'MD'), ('seek', 'VB'), ('to', 'TO'), ('draw', 'VB'), ('more', 'JJR'), ('people', 'NNS'), ('to', 'TO'), ('its', 'PRP$'), ('internet-based', 'JJ'), ('services', 'NNS'), ('with', 'IN'), ('two', 'CD'), ('new', 'JJ'), ('mid-range', 'JJ'), ('smartphones', 'NNS'), ('it', 'PRP'), ('unveiled', 'VBD'), (',', ','), ('including', 'VBG'), ('one', 'CD'), ('designed', 'VBN'), ('to', 'TO'), ('help', 'VB'), ('people', 'NNS'), ('take', 'VB'), ('better', 'JJR'), ('selfies', 'NNS'), ('.', '.')]

Modelling Tagging Problems using HMM

- We have an input sequence, that is a sequence of word tokens: $x = x_1, x_2, \dots, x_n$
- We need to determine the tag sequence for this: $y = y_1, y_2 \dots y_n$
- We can model this using HMM where we treat the input tokens to be the observable sequence and the tags to be the hidden states.
- Our goal is to determine the hidden state sequence: $y = y_1, y_2 \dots y_n$ given the input observable sequence x
- HMM will model the joint distribution: P(X, Y) where X are the input sequence and Y represents the tag sequence
- The most likely tag sequence for x is:

$$\arg\max_{y_1 \dots y_n} p(x_1 \dots x_n, y_1, y_2, \dots, y_n)$$