The Expectation Maximization (EM) Algorithm

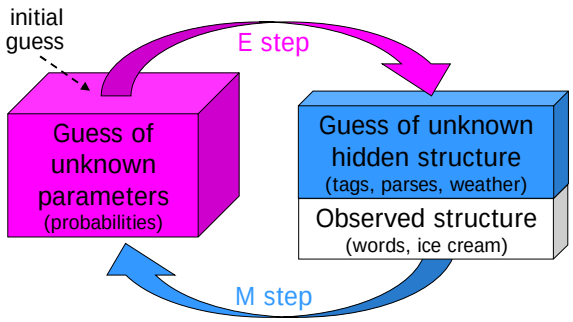
... continued!

General Idea

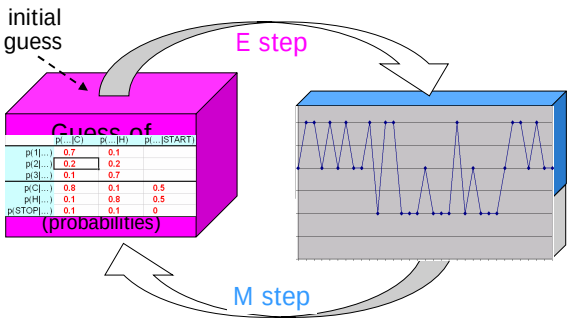
- Start by devising a noisy channel
  - Any model that predicts the corpus observations via some hidden structure (tags, parses, ...)
- Initially **guess** the parameters of the model!
  - Educated guess is best, but random can work
- **Expectation step:** Use current parameters (and observations) to reconstruct hidden structure
- **Maximization step:** Use that hidden structure (and observations) to reestimate parameters

Repeat until convergence!

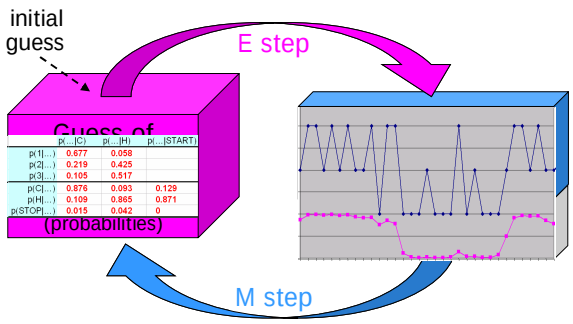
General Idea



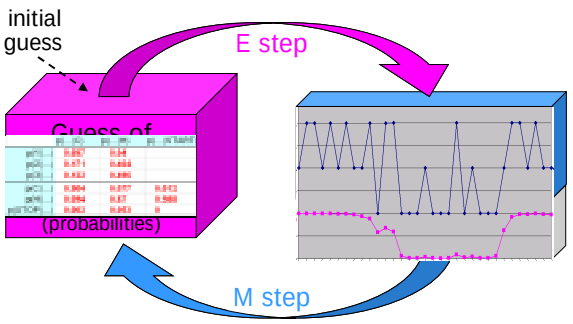
For Hidden Markov Models



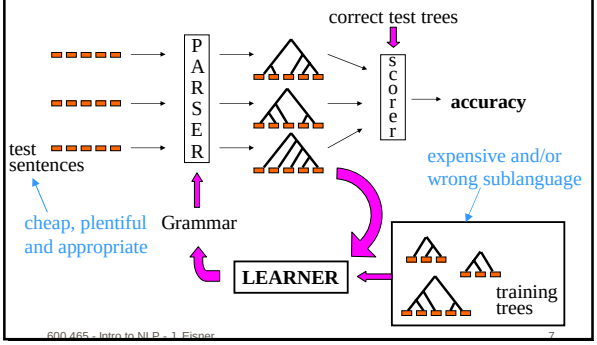
For Hidden Markov Models



For Hidden Markov Models



Grammar Reestimation



EM by Dynamic Programming: Two Versions

- The Viterbi approximation
  - Expectation: pick the *best* parse of each sentence
  - Maximization: retrain on this best-parsed corpus
  - *Advantage:* Speed!
- Real EM (why slower?)
  - Expectation: find *all* parses of each sentence
  - Maximization: retrain on *all* parses in proportion to their probability (as if we observed fractional count)
  - *Advantage:* p(training corpus) guaranteed to increase
  - Exponentially many parses, so don't extract them from chart – need some kind of clever counting

Viterbi for tagging

- *Naïve idea:* Give each word its highest-prob tag according to forward-backward.
  - Do this independently of other words.
  - **Det Adj 0.35**
  - **Det N 0.2**
  - **N V 0.45**
- Oops! We get **Det V**.
- This is defensible, but not a coherent sequence.
- Might screw up subsequent processing (parsing, chunking …) that depends on coherence.

Viterbi for tagging

- *Naïve idea:* Give each word its highest-prob tag according to forward-backward.
  - Do this independently of other words.
  - **Det Adj 0.35**
  - **Det N 0.2**
  - **N V 0.45**
- *Better idea:* Use Viterbi to pick single best tag sequence (best path).
- Can still use full EM to train the parameters, if time permits, but use Viterbi to tag once trained.

Examples

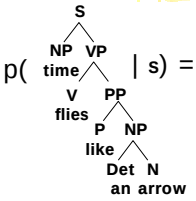
- **Finite-State** case: Hidden Markov Models
  - "forward-backward" or "Baum-Welch" algorithm
  - Applications:
    - explain ice cream in terms of underlying weather sequence
    - explain words in terms of underlying tag sequence
    - explain phoneme sequence in terms of underlying word
    - explain sound sequence in terms of underlying phoneme

  (compose these?)
- **Context-Free** case: Probabilistic CFGs
  - "inside-outside" algorithm: unsupervised grammar learning!
  - Explain raw text in terms of underlying PCFG
    - In practice, local maximum problem gets in the way
    - But *can* **improve** a good starting grammar via raw text
- **Clustering** case: explain points via clusters

Inside-Outside Algorithm

- How would we do the Viterbi approximation?
- How would we do full EM?

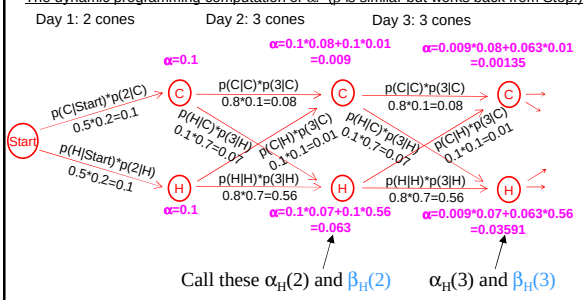
PCFG



p(| s) = p(s → NP VP | s) * p(NP → time | NP)
* p(VP → V PP | VP)
* p(V → flies | V) * …

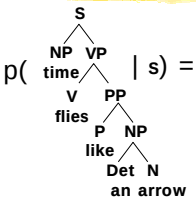
Analogies to α, β in PCFG?

The dynamic programming computation of α. (β is similar but works back from Stop.)



Call these $\alpha_H(2)$ and $\beta_H(2)$ $\alpha_H(3)$ and $\beta_H(3)$

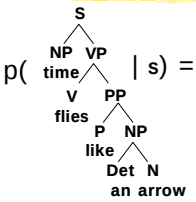
"Inside Probabilities"



p(| s) = p(s → NP VP | s) * p(NP → time | NP)
* p(VP → V PP | VP)
* p(V → flies | V) * …

- Sum over all VP parses of "flies like an arrow":
  $\beta_{VP}(1,5)$ = p(flies like an arrow | VP)
- Sum over all S parses of "time flies like an arrow":
  $\beta_S(0,5)$ = p(time flies like an arrow | S)

Compute β Bottom-Up by CKY



p(| s) = p(s → NP VP | s) * p(NP → time | NP)
* p(VP → V PP | VP)
* p(V → flies | V) * …

$\beta_{VP}(1,5)$ = p(flies like an arrow | VP) = …

$\beta_S(0,5)$ = p(time flies like an arrow | S)
= $\beta_{NP}(0,1)$ * $\beta_{VP}(1,5)$ * p(s → NP VP|s) + …

Compute β Bottom-Up by CKY

	time 1	flies 2	like 3	an 4	arrow 5
0	NP 3 Vst 3	NP 10 S 8 S 13			NP 24 NP 24 S 22 S 27 S 27
1		NP 4 VP 4			NP 18 S 21 VP 18
2			P 2 V 5		PP 12 VP 16
3				Det 1	NP 10
4					N 8

1 S → NP VP
6 S → Vst NP
2 S → S PP
1 VP → V NP
2 VP → VP PP
1 NP → Det N
2 NP → NP PP
3 NP → NP NP
0 PP → P NP

Compute β Bottom-Up by CKY

	time 1	flies 2	like 3	an 4	arrow 5
0	NP 2^{-3} Vst 2^{-3}	NP 2^{-10} S 2^{-8} S 2^{-13}			NP 2^{-24} NP 2^{-24} S 2^{-22} S 2^{-27} S 2^{-27} S 2^{-22}
1		NP 2^{-4} VP 2^{-4}			NP 2^{-18} S 2^{-21} VP 2^{-18}
2			P 2^{-2} V 2^{-5}		PP 2^{-12} VP 2^{-16}
3				Det 2^{-1}	NP 2^{-10}
4					N 2^{-8}

2^{-1} S → NP VP
2^{-6} S → Vst NP
2^{-2} S → S PP
2^{-1} VP → V NP
2^{-2} VP → VP PP
2^{-1} NP → Det N
2^{-2} NP → NP PP
2^{-3} NP → NP NP
2^{-0} PP → P NP

Compute β Bottom-Up by CKY

	time 1 flies	2 like	3 an	4 arrow	5
0	NP 2^{-3} Vst 2^{-3}	NP 2^{-10} S 2^{-8} S 2^{-13}			NP 2^{-24} NP 2^{-24} S 2^{-22} S 2^{-27} S 2^{-27} S 2^{-22} S 2^{-27}
2		NP 2^{-4} VP 2^{-4}	P 2^{-2} V 2^{-5}		NP 2^{-18} S 2^{-21} VP 2^{-18} PP 2^{-12} VP 2^{-16}
3				Det 2^{-1}	NP 2^{-10}
4					N 2^{-8}

2^{-1} S → NP VP
2^{-6} S → Vst NP
2^{-2} S → S PP
2^{-1} VP → V NP
2^{-2} VP → VP PP
2^{-1} NP → Det N
2^{-2} NP → NP PP
2^{-3} NP → NP NP
2^{-0} PP → P NP

The Efficient Version: Add as we go

	time 1 flies	2 like	3 an	4 arrow	5
0	NP 2^{-3} Vst 2^{-3}	NP 2^{-10} S 2^{-8} S 2^{-13}			NP 2^{-24} NP 2^{-24} S 2^{-22} S 2^{-27} S 2^{-27}
2		NP 2^{-4} VP 2^{-4}	P 2^{-2} V 2^{-5}		NP 2^{-18} S 2^{-21} VP 2^{-18} PP 2^{-12} VP 2^{-16}
3				Det 2^{-1}	NP 2^{-10}
4					N 2^{-8}

2^{-1} S → NP VP
2^{-6} S → Vst NP
2^{-2} S → S PP
2^{-1} VP → V NP
2^{-2} VP → VP PP
2^{-1} NP → Det N
2^{-2} NP → NP PP
2^{-3} NP → NP NP
2^{-0} PP → P NP

The Efficient Version: Add as we go

	time 1 flies	2 like	3 an	4 arrow	5
0	NP 2^{-3} Vst 2^{-3}	NP 2^{-10} S 2^{-8} +2^{-13}			NP 2^{-24} +2^{-24} S 2^{-22} +2^{-27} +2^{-27}
2		NP 2^{-4} VP 2^{-4}	P 2^{-2} V 2^{-5}		NP 2^{-18} S 2^{-21} VP 2^{-18} PP 2^{-12} VP 2^{-16}
3				Det 2^{-1}	NP 2^{-10}
4					N 2^{-8}

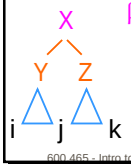
$\beta_S(0,2)$ $\beta_S(0,5)$
$\beta_S(0,2) * \beta_{PP}(2,5) * p(S \rightarrow S\ PP \mid S)$
+2^{-22}
+2^{-27}
$\beta_{PP}(2,5)$

2^{-1} S → NP VP
2^{-6} S → Vst NP
2^{-2} S → S PP
2^{-1} VP → V NP
2^{-2} VP → VP PP
2^{-1} NP → Det N
2^{-2} NP → NP PP
2^{-3} NP → NP NP
2^{-0} PP → P NP

Compute β probs bottom-up (CKY)

need some initialization up here for the width-1 case

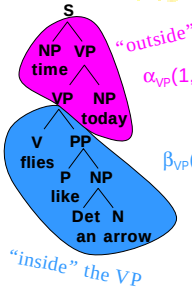
```
for width := 2 to n                    (* build smallest first *)
  for i := 0 to n-width                (* start *)
    let k := i + width                 (* end *)
    for j := i+1 to k-1                (* middle *)
      for all grammar rules X → Y Z
        βX(i,k) += p(X → Y Z | X) * βY(i,j) * βZ(j,k)
```



what if you changed + to max?

what if you replaced all rule probabilities by 1?

Inside & Outside Probabilities



"outside" the VP

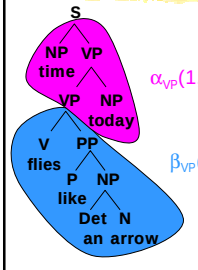
"inside" the VP

$\alpha_{VP}(1,5)$ = p(time VP today | S)

$\beta_{VP}(1,5)$ = p(flies like an arrow | VP)

$\alpha_{VP}(1,5) * \beta_{VP}(1,5)$
= p(time [VP flies like an arrow] today | S)

Inside & Outside Probabilities



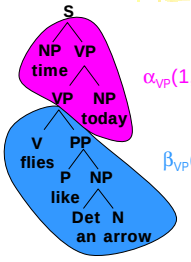
$\alpha_{VP}(1,5)$ = p(time VP today | S)

$\beta_{VP}(1,5)$ = p(flies like an arrow | VP)

$\alpha_{VP}(1,5) * \beta_{VP}(1,5) / \beta_S(0,6)$
$= \frac{p(\text{time flies like an arrow today \& VP(1,5)} \mid S)}{p(\text{time flies like an arrow today} \mid S)}$

= p(VP(1,5) | time flies like an arrow today, S)

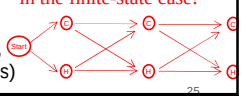
Inside & Outside Probabilities



$\alpha_{VP}(1,5)$ = p(time VP today | S)

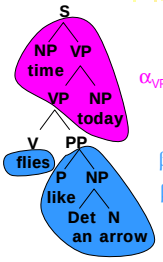
$\beta_{VP}(1,5)$ = p(flies like an arrow | VP)

strictly analogous to forward-backward in the finite-state case!



So $\alpha_{VP}(1,5) * \beta_{VP}(1,5) / \beta_s(0,6)$
is probability that there is a VP here,
given all of the observed data (words)

Inside & Outside Probabilities



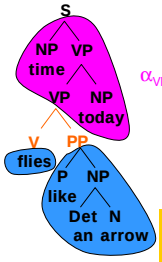
$\alpha_{VP}(1,5)$ = p(time VP today | S)

$\beta_V(1,2)$ = p(flies | V)

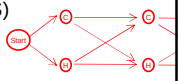
$\beta_{PP}(2,5)$ = p(like an arrow | PP)

So $\alpha_{VP}(1,5) * \beta_V(1,2) * \beta_{PP}(2,5) / \beta_s(0,6)$
is probability that there is VP → V PP here,
given all of the observed data (words) *... or is it?*

Inside & Outside Probabilities



strictly analogous to forward-backward in the finite-state case!



$\alpha_{VP}(1,5)$ = p(time VP today | S)

$\beta_V(1,2)$ = p(flies | V)

$\beta_{PP}(2,5)$ = p(like an arrow | PP)

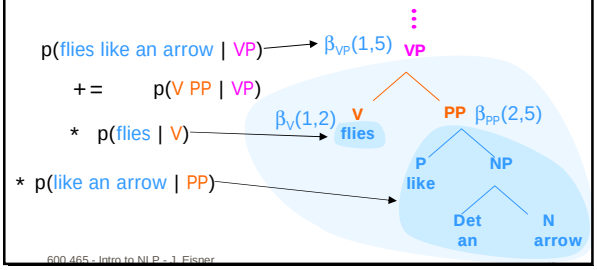
sum prob over all position triples like (1,2,5) to get expected c(VP → V PP); reestimate PCFG!

So $\alpha_{VP}(1,5) * p(VP \rightarrow V\ PP) * \beta_V(1,2) * \beta_{PP}(2,5) / \beta_s(0,6)$
is probability that there is VP → V PP here (at 1-2-5),
given all of the observed data (words)

Compute β probs bottom-up

Summary: $\beta_{VP}(1,5)$ += p(V PP | VP) * $\beta_V(1,2)$ * $\beta_{PP}(2,5)$

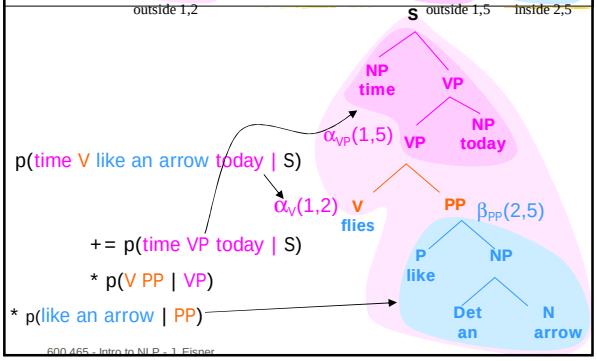
inside 1,5 inside 1,2 inside 2,5



Compute α probs top-down

Summary: $\alpha_V(1,2)$ += p(V PP | VP) * $\alpha_{VP}(1,5)$ * $\beta_{PP}(2,5)$

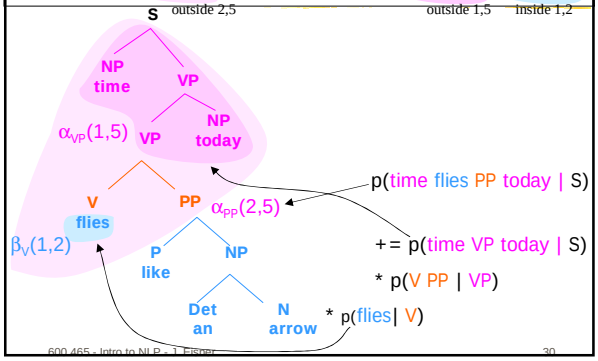
outside 1,2 outside 1,5 inside 2,5



Compute α probs top-down

Summary: $\alpha_{PP}(2,5)$ += p(V PP | VP) * $\alpha_{VP}(1,5)$ * $\beta_V(1,2)$

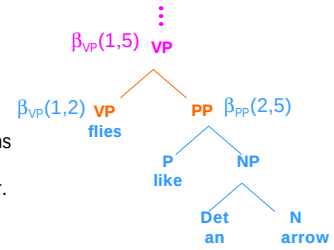
outside 2,5 outside 1,5 inside 1,2



Compute β probs bottom-up

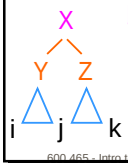
When you build VP(1,5), from VP(1,2) and VP(2,5) during CKY, increment $\beta_{VP}(1,5)$ by
p(VP → VP PP) *
$\beta_{VP}(1,2)$ * $\beta_{PP}(2,5)$

Why? $\beta_{VP}(1,5)$ is total probability of all derivations p(flies like an arrow | VP) and we just found another. (See earlier slide of CKY chart.)



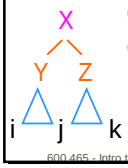
Compute β probs bottom-up (CKY)

```
for width := 2 to n                    (* build smallest first *)
  for i := 0 to n-width                (* start *)
    let k := i + width                 (* end *)
    for j := i+1 to k-1                (* middle *)
      for all grammar rules X → Y Z
        βX(i,k) += p(X → Y Z) * βY(i,j) * βZ(j,k)
```



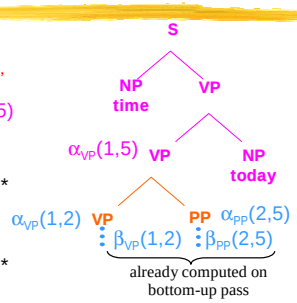
Compute α probs top-down (reverse CKY)

```
for width := 2 to n  (n downto 2)      "unbuild" biggest first
                                       (* build smallest first *)
  for i := 0 to n-width                (* start *)
    let k := i + width                 (* end *)
    for j := i+1 to k-1                (* middle *)
      for all grammar rules X → Y Z
        αY(i,j) += ???
        αZ(j,k) += ???
```



Compute α probs top-down

After computing β during CKY, revisit constits in reverse order (i.e., bigger constituents first).
When you "unbuild" VP(1,5) from VP(1,2) and VP(2,5), increment $\alpha_{VP}(1,2)$ by
$\alpha_{VP}(1,5)$ * p(VP → VP PP) * $\beta_{PP}(2,5)$
and increment $\alpha_{PP}(2,5)$ by
$\alpha_{VP}(1,5)$ * p(VP → VP PP) * $\beta_{VP}(1,2)$



$\alpha_{VP}(1,2)$ is total prob of all ways to gen VP(1,2) and all outside words.

Compute α probs top-down (reverse CKY)

```
for width := 2 to n  (n downto 2)      "unbuild" biggest first
                                       (* build smallest first *)
  for i := 0 to n-width                (* start *)
    let k := i + width                 (* end *)
    for j := i+1 to k-1                (* middle *)
      for all grammar rules X → Y Z
        αY(i,j) += αX(i,k) * p(X → Y Z) * βZ(j,k)
        αZ(j,k) += αX(i,k) * p(X → Y Z) * βY(i,j)
```