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

initial guess

- Guess of unknown hidden structure (tags, parses, weather)
- Observed structure (words, ice cream)

Guess of unknown parameters (probabilities)

E step

M step

For Hidden Markov Models

initial guess

- Guess of unknown hidden structure (tags, parses, weather)
- Observed structure (words, ice cream)

Guess of unknown parameters (probabilities)

E step

M step

For Hidden Markov Models

initial guess

- Guess of unknown hidden structure (tags, parses, weather)
- Observed structure (words, ice cream)

Guess of unknown parameters (probabilities)

E step

M step
Grammar Reestimation

- **E step**
  - Correct test trees
  - Accuracy

- **M step**
  - Training trees
  - Grammar reestimator

Grammar Reestimation

- **Grammar**
  - Cheap, plentiful and appropriate
  - Expensive and/or wrong sublanguage

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**
    - **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

Examples of EM

- **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 phoneme
- **Context-Free** case: Probabilistic CFGs
  - "inside-outside" algorithm: unsupervised grammar learning!
  - Explain raw text in terms of underlying cx-free parse
  - 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

Our old friend PCFG

- **S**
  - NP | VP | time | V | PRT
  - P | N
  - like
  - Det | N
  - an arrow

Viterbi reestimation for parsing

- Start with a "pretty good" grammar
  - E.g., it was trained on supervised data (a treebank) that is small, imperfectly annotated, or has sentences in a different style from what you want to parse.
- Parse a corpus of unparsed sentences:
  - # copies of this sentence in the corpus
  - Reestimate:
    - Collect counts: \( c(S \rightarrow NP \ VP) \) += 12; \( c(S) \) += 2*12; ...
    - Divide: \( p(S \rightarrow NP \ VP \ | \ S) = c(S \rightarrow NP \ VP) / c(S) \)
    - May be wise to smooth

True EM for parsing

- Similar, but now we consider all parses of each sentence
- Parse our corpus of unparsed sentences:
  - # copies of this sentence in the corpus
  - Collect counts fractionally
    - \( c(S \rightarrow NP \ VP) \) += 10.8; \( c(S) \) += 2*10.8; ...
    - \( c(S \rightarrow NP \ VP) \) += 1.2; \( c(S) \) += 1*1.2; ...
  - But there may be exponentially many parses of a length-n sentence!
  - How can we stay fast? Similar to taggings...
**Analogies to α, β in PCFG?**

The dynamic programming computation of \( \alpha \) is similar but works back from Stop:

- Day 1: 2 cones
- Day 2: 3 cones
- Day 3: 3 cones

Compute

\[
\begin{align*}
\alpha_S(0,5) &= p(s) \cdot p(NP \rightarrow time) \cdot p(NP) \\
&= 0.5 \cdot 0.2 = 0.1
\end{align*}
\]

**“Inside Probabilities”**

\[
\begin{align*}
p(s \rightarrow NP \; VP \; s) &= 0.5 \\
p(NP \rightarrow time \; NP) &= 0.2 \\
p(VP \rightarrow V \; PP) &= 0.1 \\
p(V \rightarrow V \; PP) &= 0.1
\end{align*}
\]

- Sum over all VP parses of “flies like an arrow”:
  \( \beta_{VP}(1,5) = p(\text{flies like an arrow} \; | \; VP) \)
- Sum over all S parses of “time flies like an arrow”:
  \( \beta_{S}(0,5) = p(\text{time flies like an arrow} \; | \; s) \)

**Compute β Bottom-Up by CKY**

\[
\begin{align*}
\beta_{NP}(0,1) &= p(s \rightarrow NP \; VP) \\
\beta_{VP}(1,5) &= p(\text{flies like an arrow} \; | \; VP) \\
\beta_{S}(0,5) &= p(\text{time flies like an arrow} \; | \; s)
\end{align*}
\]

Call these \( \alpha(2) \) and \( \beta(2) \), \( \alpha(3) \) and \( \beta(3) \)

**Compute β Bottom-Up by CKY**

\[
\begin{align*}
p(s \rightarrow NP \; VP \; s) &= 0.5 \\
p(NP \rightarrow time \; NP) &= 0.2 \\
p(VP \rightarrow V \; PP) &= 0.1 \\
p(V \rightarrow V \; PP) &= 0.1
\end{align*}
\]

- Sum over all VP parses of “flies like an arrow”:
  \( \beta_{VP}(1,5) = p(\text{flies like an arrow} \; | \; VP) \)
- Sum over all S parses of “time flies like an arrow”:
  \( \beta_{S}(0,5) = p(\text{time flies like an arrow} \; | \; s) \)

**Compute β Bottom-Up by CKY**

\[
\begin{align*}
\beta_{NP}(0,1) &= p(s \rightarrow NP \; VP) \\
\beta_{VP}(1,5) &= p(\text{flies like an arrow} \; | \; VP) \\
\beta_{S}(0,5) &= p(\text{time flies like an arrow} \; | \; s)
\end{align*}
\]

Call these \( \alpha(2) \) and \( \beta(2) \), \( \alpha(3) \) and \( \beta(3) \)
The Efficient Version: Add as we go

<table>
<thead>
<tr>
<th>time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>2^3</td>
<td>NP</td>
<td>NP</td>
<td>2^10</td>
<td>2^4</td>
</tr>
<tr>
<td>Vst</td>
<td>2^3</td>
<td>S</td>
<td>2^4</td>
<td>2^13</td>
<td>2^19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S</td>
<td>2^17</td>
<td>2^19</td>
<td></td>
</tr>
</tbody>
</table>

Compute \( \beta \) 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 \( X \to Y Z \)
        \( \beta_x(i,k) += p(X \to Y Z | X) \cdot \beta_y(i,j) \cdot \beta_z(j,k) \)

Inside & Outside Probabilities

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

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

Strictly analogous to forward-backward in the finite-state case!
Inside & Outside Probabilities

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

\[ \beta_{(1,2)} = p(\text{flies} | V) \]
\[ \beta_{(2,5)} = p(\text{like an arrow} | \text{PP}) \]

So \( \alpha_{(1,5)} \cdot \beta_{(1,2)} \cdot \beta_{(2,5)} / \beta_{(0,6)} \)

is probability that there is VP \( V \) PP here, given all of the observed data (words) ... or is it?

Compute \( \beta \) probs bottom-up

(gradually build up larger blue “inside” regions)

Summary:
\[ \beta_{(1,5)} \] += \( p(V \text{ PP} | V) \) * \( \beta_{(1,2)} \) * \( \beta_{(2,5)} \)

\[ p(\text{flies like an arrow} | V) \rightarrow \beta_{(1,5)} \] VP

\[ \beta_{(1,2)} \] * \( p(\text{flies} | V) \)

* \( p(\text{like an arrow} | \text{PP}) \)

Compute \( \alpha \) probs top-down

Summary:
\[ \alpha_{(2,5)} \] += \( p(V \text{ PP} | V) \) * \( \alpha_{(1,5)} \) * \( \beta_{(1,2)} \)

\[ p(\text{time flies PP today} | S) \rightarrow \alpha_{(2,5)} \] VP

\[ \beta_{(1,2)} \] * \( p(\text{flies} | V) \)

* \( p(\text{like an arrow} | \text{PP}) \)

Compute \( \beta \) probs bottom-up

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

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

Details:

Compute \( \beta \) probs bottom-up

Inside & Outside Probabilities

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

So \( \alpha_{(1,5)} \cdot p(\text{VP} \rightarrow V \text{ PP}) \cdot \beta_{(1,2)} \cdot \beta_{(2,5)} / \beta_{(0,6)} \)

is probability that there is VP \( V \) PP here (at 1-2-5), given all of the observed data (words)

\[ \sum \text{prob over all position triples like (1,2,5)} \]

get expected \( c(VP V \text{ PP}) \); reestimate PCFG!

Compute \( \alpha \) probs top-down

Summary:
\[ \alpha_{(1,2)} \] += \( p(V \text{ PP} | V) \) * \( \alpha_{(1,5)} \) * \( \beta_{(2,5)} \)

\[ p(\text{time flies PP today} | S) \rightarrow \alpha_{(1,2)} \] VP

\[ \beta_{(1,2)} \] * \( p(\text{flies} | V) \)

* \( p(\text{like an arrow} | \text{PP}) \)

\[ \alpha_{(2,5)} \] += \( p(V \text{ PP} | V) \) * \( \alpha_{(1,5)} \) * \( \beta_{(1,2)} \)

\[ p(\text{time flies PP today} | S) \rightarrow \alpha_{(2,5)} \] VP

\[ \beta_{(1,2)} \] * \( p(\text{flies} | V) \)

* \( p(\text{like an arrow} | \text{PP}) \)
Details: Compute $\beta$ probs bottom-up (CKY)

for width := 2 to n
(*) build smallest first (*)
for i := 0 to n-width
let k := i + width
(*) end (*)
for j := i+1 to k-1
(*) middle (*)
for all grammar rules $X \rightarrow Y Z$
\[
\beta_x(i,k) += p(X \rightarrow Y Z) \cdot \beta_y(i,j) \cdot \beta_z(j,k)
\]

Details: Compute $\alpha$ probs top-down (reverse 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 $X \rightarrow Y Z$
\[
\alpha_x(i,j) += ???
\]
\[
\alpha_y(j,k) += ???
\]

Details: Compute $\alpha$ probs top-down (reverse CKY)

After computing $\beta$ during CKY, revisit constituents in reverse order (i.e., biggest constituents first). When you "unbuild" VP(1,5) from VP(1,2) and VP(2,5), increment $\alpha_{vp}(1,2)$ by $\alpha_{vp}(1,5) \cdot p(VP \rightarrow VP PP) \cdot \beta_{vp}(2,5)$ and increment $\alpha_{vp}(2,5)$ by $\alpha_{vp}(1,5) \cdot p(VP \rightarrow VP PP) \cdot \beta_{vp}(1,2)$

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

What Inside-Outside is Good For

1. As the E step in the EM training algorithm
2. Predicting which nonterminals are probably where
3. Viterbi version as an A* or pruning heuristic
4. As a subroutine within non-context-free models

What Inside-Outside is Good For

1. As the E step in the EM training algorithm
   - That's why we just did it

\[
c(S) += \sum_{i,j} \alpha_x(i,j) \cdot \beta_x(i,j)/Z
\]
\[
c(S \rightarrow NP VP) += \sum_{i,j,k} \alpha_y(i,k) \cdot p(S \rightarrow NP VP) \cdot \beta_{vp}(i,j,k)/Z
\]
where $Z$ = total prob of all parses = $\beta_x(0,n)$
What Inside-Outside is Good For

1. As the E step in the EM training algorithm
2. Predicting which nonterminals are probably where
   - Posterior decoding of a single sentence
     - Like using v to pick the most probable tag for each word
     - But can't just pick most probable nonterminal for each span ...
     - Wouldn't get a tree! (Not all spans are constituents.)
   - So, find the tree that maximizes expected # correct nonterms.
   - Alternatively, expected # of correct rules.
   - For each nonterminal (or rule), at each position:
     - v tells you the probability that it's correct.
     - For a given tree, sum these probabilities over all positions to get that tree's expected # of correct nonterminals (or rules).
   - How can we find the tree that maximizes this sum?
     - Dynamic programming – just weighted CKY all over again.
     - But now the weights come from v instead of inside-outside.

What Inside-Outside is Good For

1. As the E step in the EM training algorithm
2. Predicting which nonterminals are probably where
   - As soft features in a predictive classifier
   - Pruning the parse forest of a sentence
     - To build a packed forest of all parse trees, keep all backpointer pairs
     - But can't just pick most probable nonterminal for each span …
     - So, find the tree that maximizes expected # correct nonterms.
     - For each nonterminal (or rule), at each position:
       - v tells you the probability that it's correct.
       - For a given tree, sum these probabilities over all positions to get that tree's expected # of correct nonterminals (or rules).
     - How can we find the tree that maximizes this sum?
       - Dynamic programming – just weighted CKY all over again.
       - But now the weights come from v instead of inside-outside.

What Inside-Outside is Good For

1. As the E step in the EM training algorithm
2. Predicting which nonterminals are probably where
3. Viterbi version as an A* or pruning heuristic
   - Viterbi inside-outside uses a semiring with max in place of +
   - Call the resulting quantities v, instead of \(p_{\text{Viterbi}}\) (as for HMM)

Prob of best parse that contains a constituent \(x\) is \(\mu(x)\), and all other constituents have \(\mu(x) < p\).

So if we only know \(v(x) < p\), we could skip working on \(x\).

If you're sure it's an NP, the feature fires

If you're sure it's not an NP, the feature doesn't fire

You want to predict whether the substring from \(i\) to \(j\) is a name

As soft features in a predictive classifier

As the E step in the EM training algorithm

As soft features in a predictive classifier

As the E step in the EM training algorithm

As the E step in the EM training algorithm

As the E step in the EM training algorithm

Viterbi inside-outside uses a semiring with max in place of +

Call the resulting quantities \(v\), instead of \(p_{\text{Viterbi}}\) (as for HMM)

Prob of best parse that contains a constituent \(x\) is \(v(x)\), and all other constituents have \(v(x) < p\).

So if we only know \(v(x) < p\), we could skip working on \(x\).

In the parsing tricks lecture, we wanted to prioritize or prune \(x\) according to \(\mu(x)\). We now see better what \(q(x)\) was:

\(q(x) = v(x)\) was just the Viterbi inside probability: \(v(x) = v(x)\)

\(q(x)\) was just an estimate of the Viterbi outside prob: \(q(x) < v(x)\)

\(q(x)\) is just an estimate of the Viterbi outside prob: \(q(x) = v(x)\).

But to compute \(\mu(x)\) exactly, prioritization would first process the constituents with maximum \(v(x)\), which are just the correct ones! So we would do unecessary work.

But if we can guarantee \(q(x) < v(x)\), get a safe A* algorithm.

We can find such \(q(x)\) values by first running Viterbi inside-outside on the sentence using a faster, approximate grammar …

...
What Inside-Outside is Good For

1. As the E step in the EM training algorithm
2. Predicting which nonterminals are probably where
3. Viterbi version as an A* or pruning heuristic
4. As a subroutine within non-context-free models
   - We’ve always defined the weight of a parse tree as the sum of its rules’ weights.
   - Advanced topic: Can do better by considering additional features of the tree (“non-local features”), e.g., within a log-linear model.
   - CKY no longer works for finding the best parse, ☹
     - Approximate “reranking” algorithm: Using a simplified model that uses only local features, use CKY to find a parse forest. Extract the best 1000 parses. Then re-score these 1000 parses using the full model.
     - Better approximate and exact algorithms: Beyond scope of this course. But they usually call inside-outside or Viterbi inside-outside as a subroutine, often several times (on multiple variants of the grammar, where again each variant can only use local features).