Structured Prediction with Perceptrons and CRFs

Time flies like an arrow

Time flies like an arrow

Time flies like an arrow

Time flies like an arrow

...
Structured Prediction with Perceptrons and CRFs

Back to conditional log-linear modeling ...

But now, model structures!
\[ p(\text{fill} \mid \text{shape}) \]

<table>
<thead>
<tr>
<th>Solid</th>
<th>Striped</th>
<th>Hollow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Triangle &amp; Solid</td>
<td>Triangle &amp; Striped</td>
<td>Triangle &amp; Hollow</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Circle &amp; Solid</td>
<td>Circle &amp; Striped</td>
<td>Circle &amp; Hollow</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pentagon &amp; Solid</td>
<td>Pentagon &amp; Striped</td>
<td>Pentagon &amp; Hollow</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Context**

\[ N_{\text{triangle}} = \frac{15}{5} \]

\[ N_{\text{circle}} = \frac{100}{70} \]

\[ N_{\text{pentagon}} = \frac{0}{0} \]
$p(\text{fill} \mid \text{shape})$

<table>
<thead>
<tr>
<th>solid</th>
<th>striped</th>
<th>hollow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3052</td>
<td>-0.18e-2</td>
<td>-2.2671</td>
</tr>
</tbody>
</table>

- Triangle & solid: -0.9454
- Triangle & striped: 0.496-2
- Triangle & hollow: 0.2820
- Circle & solid: 0.7036
- Circle & striped: 0.1236
- Circle & hollow: 0.628
- Pentagon & solid: 0
- Pentagon & striped: 0
- Pentagon & hollow: 0

**Context**

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Circle</th>
<th>Pentagon</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 [N_{\text{triangle}} = 15]</td>
<td>70 [N_{\text{circle}} = 100]</td>
<td>0 [N_{\text{pentagon}} = 0]</td>
</tr>
</tbody>
</table>
$p(\text{category} \mid \text{message})$

<table>
<thead>
<tr>
<th>Goodmail</th>
<th>Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reply today to claim your ...</td>
<td>Reply today to claim your ...</td>
</tr>
<tr>
<td>Wanna get pizza tonight?</td>
<td>Wanna get pizza tonight?</td>
</tr>
<tr>
<td>Thx; consider enlarging the ...</td>
<td>Thx; consider enlarging the ...</td>
</tr>
<tr>
<td>Enlarge your hidden ...</td>
<td>Enlarge your hidden ...</td>
</tr>
</tbody>
</table>
\[ p(\text{RHS} \mid \text{LHS}) \]

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3582</td>
<td></td>
<td>-6.10e-2</td>
<td></td>
<td>-0.2971</td>
</tr>
<tr>
<td>-0.3454</td>
<td></td>
<td>6.46e-2</td>
<td></td>
<td>0.2809</td>
</tr>
<tr>
<td>0.7036</td>
<td></td>
<td>-0.1256</td>
<td></td>
<td>-0.578</td>
</tr>
</tbody>
</table>

\[ S \rightarrow \text{NP} \ \text{VP} \]
\[ S \rightarrow \text{N} \ \text{VP} \]
\[ S \rightarrow \text{NP}[+\text{wh}] \ \text{V} \ \text{S/V/NP} \]
\[ S \rightarrow \text{VP} \ \text{NP} \]
\[ S \rightarrow \text{Det} \ \text{N} \]
\[ S \rightarrow \text{PP} \ \text{P} \]

\[ \ldots \]
p(RHS | LHS)

\[
\begin{align*}
S & \rightarrow \text{NP VP} & S & \rightarrow \text{N VP} & S & \rightarrow \text{NP[+wh] V S/ V/ NP} \\
S & \rightarrow \text{VP NP} & S & \rightarrow \text{Det N} & S & \rightarrow \text{PP P} \\
\text{NP} & \rightarrow \text{NP VP} & \text{NP} & \rightarrow \text{N VP} & \text{NP} & \rightarrow \text{NP CP/ NP} \\
\text{NP} & \rightarrow \text{VP NP} & \text{NP} & \rightarrow \text{Det N} & \text{NP} & \rightarrow \text{NP PP}
\end{align*}
\]
\[ p(\text{parse} \mid \text{sentence}) \]

Time flies like an arrow

\[
\begin{array}{c}
S \\
| \quad \text{VP} \\
| \quad \text{PP} \\
| \quad \text{NP} \\
N \\
V \\
P \\
D \\
N
\end{array}
\]

Time flies like an arrow

\[
\begin{array}{c}
S \\
| \quad \text{VP} \\
| \quad \text{PP} \\
V \\
N \\
P \\
D \\
N
\end{array}
\]

Time flies like an arrow

\[
\begin{array}{c}
S \\
| \quad \text{VP} \\
| \quad \text{NP} \\
N \\
N \\
V \\
D \\
N
\end{array}
\]

Time flies like an arrow

\[
\begin{array}{c}
S \\
| \quad \text{VP} \\
| \quad \text{NP} \\
V \\
V \\
V \\
D \\
N
\end{array}
\]

Time flies like an arrow
$p(\text{tag sequence} \mid \text{word sequence})$

Time flies like an arrow

\[
\begin{array}{cccccc}
N & V & P & D & N & N \\
\text{Time flies like an arrow} & V & N & P & D & N \\
\end{array}
\]

\[
\begin{array}{cccccc}
N & N & V & D & N & N \\
\text{Time flies like an arrow} & V & V & V & D & N \\
\end{array}
\]

\[
\begin{array}{cccccc}
N & V & P & D & N & N \\
\text{Time flies like an arrow} & V & N & P & D & N \\
\end{array}
\]
Today’s general problem

- Given some input $x$
  - Occasionally empty, e.g., no input needed for a generative n-gram or model of strings (randsent)

- Consider a set of candidate outputs $y$
  - Classifications for $x$ (small number: often just 2)
  - Taggings of $x$ (exponentially many)
  - Parses of $x$ (exponential, even infinite)
  - Translations of $x$ (exponential, even infinite)
  - ...

- Want to find the “best” $y$, given $x$
Remember Weighted CKY ... (find the minimum-weight parse)

time  1  flies  2  like  3  an  4  arrow  5

<table>
<thead>
<tr>
<th></th>
<th>NP  3</th>
<th>NP  10</th>
<th></th>
<th>NP  24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vst  3</td>
<td>S</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>NP  4</td>
<td>NP  18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VP  4</td>
<td>S</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P</td>
<td>2</td>
<td>PP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>5</td>
<td>VP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Det</td>
<td>1</td>
<td>NP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>N</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

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
So far, we used weighted CKY only to implement probabilistic CKY for PCFGs

But is weighted CKY good for anything else??

<table>
<thead>
<tr>
<th>time</th>
<th>1 flies</th>
<th>2 like</th>
<th>3 an</th>
<th>4 arrow</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NP 3</td>
<td>NP 10</td>
<td></td>
<td>NP 24</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vst 3</td>
<td>S 8</td>
<td></td>
<td>S 22</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>NP 4</td>
<td>VP 4</td>
<td>2^8</td>
<td>NP 18</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S 21</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VP 18</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P 2</td>
<td>VP 2</td>
<td>2^12</td>
<td>PP 12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>V 5</td>
<td></td>
<td></td>
<td>VP 16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Det 1</td>
<td>NP 10</td>
<td></td>
<td>Det 1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>N 8</td>
<td></td>
<td>N 8</td>
<td></td>
</tr>
</tbody>
</table>

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

multiply to get 2^-22
We set the weights to log probs

\[ w(S) = w(S \rightarrow NP \ VP) + w(NP \rightarrow time) \]
\[ + w(\text{VP} \rightarrow \text{VP} \ NP) \]
\[ + w(\text{VP} \rightarrow \text{flies}) + \ldots \]

Just let \( w(X \rightarrow Y \ Z) = -\log p(X \rightarrow Y \ Z \mid X) \)

Then lightest tree has highest prob

But is weighted CKY good for anything else?? Do the weights have to be probabilities?
Probability is Useful

- We love probability distributions!
  - We’ve learned how to define & **use** \(p(\ldots)\) functions.
- Pick best output text \(T\) from a set of candidates
  - speech recognition (HW2); machine translation; OCR; spell correction...
  - maximize \(p_1(T)\) for some appropriate distribution \(p_1\)
- Pick best annotation \(T\) for a fixed input \(I\)
  - text categorization; parsing; part-of-speech tagging ...
  - maximize \(p(T \mid I)\); equivalently maximize joint probability \(p(I,T)\)
    - often define \(p(I,T)\) by noisy channel: \(p(I,T) = p(T) \times p(I \mid T)\)
  - speech recognition & other tasks above are cases of this too:
    - we’re maximizing an appropriate \(p_1(T)\) defined by \(p(T \mid I)\)
- Pick best probability distribution (a meta-problem!)
  - really, pick best **parameters** \(\theta\): train HMM, PCFG, n-grams, clusters ...
  - maximum likelihood; smoothing; EM if unsupervised (incomplete data)
  - Bayesian smoothing: \(\max p(\theta \mid \text{data}) = \max p(\theta, \text{data}) = p(\theta)p(\text{data} \mid \theta)\)
Probability is Flexible

- We love probability distributions!
  - We’ve learned how to define & use $p(\cdot)$ functions.
- We want $p(\cdot)$ to define probability of linguistic objects
  - Trees of (non)terminals (PCFGs; CKY, Earley, pruning, inside-outside)
  - Sequences of words, tags, morphemes, phonemes ($n$-grams, FSAs, FSTs; regex compilation, best-paths, forward-backward, collocations)
  - Vectors (decis.lists, Gaussians, naïve Bayes; Yarowsky, clustering/k-NN)
- We’ve also seen some not-so-probabilistic stuff
  - Syntactic features, semantics, morph., Gold. Could be stochasticized?
  - Methods can be quantitative & data-driven but not fully probabilistic: transf.-based learning, bottom-up clustering, LSA, competitive linking
- But probabilities have wormed their way into most things
- $p(\cdot)$ has to capture our intuitions about the ling. data

summary of other half of the course (linguistics)
An Alternative Tradition

- Old AI hacking technique:
  - Possible parses (or whatever) have scores.
  - Pick the one with the best score.
  - How do you define the score?
    - Completely ad hoc!
    - **Throw anything you want into the stew**
    - Add a bonus for this, a penalty for that, etc.
Scoring by Linear Models

- Given some input \( x \)
- Consider a set of candidate outputs \( y \)
- Define a scoring function \( \text{score}(x, y) \)
  
  Linear function: A sum of feature weights (you pick the features!)

  \[
  \text{score}(x, y) = \sum_k \theta_k f_k(x, y)
  \]

  Weight of feature \( k \)
  (learned or set by hand)

  Ranges over all features, e.g., \( k=5 \) (numbered features)
  or \( k=\text{"see Det Noun"} \) (named features)

  Whether \( (x, y) \) has feature \( k \) (0 or 1)
  Or how many times it fires (\( \geq 0 \))
  Or how strongly it fires (real #)

- Choose \( y \) that maximizes \( \text{score}(x, y) \)
Scoring by Linear Models

- Given some input $x$
- Consider a set of candidate outputs $y$
- Define a scoring function $\text{score}(x, y)$
  
  Linear function: A sum of feature weights (you pick the features!)

$\begin{align*}
\text{score}(x, y) &= \theta \cdot f(x, y)
\end{align*}$

(learned or set by hand)

This linear decision rule is sometimes called a “perceptron.”
It’s a “structured perceptron” if it does structured prediction
(number of $y$ candidates is unbounded, e.g., grows with $|x|$).

- Choose $y$ that maximizes $\text{score}(x, y)$
Related older ideas

- Linear regression predicts a real number $y$:
  \[ y = \text{prediction}(x) = \theta \cdot f(x) \]

- Binary classification:
  \[ \text{score}(x, \text{spam}) = \theta \cdot f(x) \]
  \[ \text{score}(x, \text{gen}) = 0 \]
  - Predict “spam” if $\theta \cdot f(x) > 0$

- Our multi-class classification uses $f(x,y)$, not $f(x)$:
  \[ \text{score}(x, y) = \theta \cdot f(x, y) \]
  - Predict $y$ that maximizes $\text{score}(x,y)$
  - If only 2 possible values of $y$, equivalent to binary case
An Alternative Tradition

- Old AI hacking technique:
  - Possible parses (or whatever) have scores.
  - Pick the one with the best score.
  - How do you define the score?
    - Completely ad hoc!
    - *Throw anything you want into the stew*
      - Add a bonus for this, a penalty for that, etc.
  - “Learns” over time - as you adjust bonuses and penalties by hand to improve performance.
  - Total kludge, but totally flexible too ...
    - Can throw in *any* intuitions you might have
  - Could we make it learn automatically?
Perceptron Training Algorithm

- initialize $\theta$ (usually to the zero vector)
- repeat:
  - Pick a training example $(x,y)$
  - Model predicts $\hat{y}$ that maximizes score$(x,\hat{y})$
  - Update weights by a step of size $\epsilon > 0$:
    $$\theta = \theta + \epsilon \cdot (f(x,y) - f(x,\hat{y}))$$

If model prediction was correct ($y=\hat{y}$), $\theta$ doesn’t change. So once model predicts all training examples correctly, stop.

If some $\theta$ can do the job, this eventually happens!
(If not, $\theta$ will oscillate, but the average $\theta$ from all steps will settle down. So return that eventual average.)
Perceptron Training Algorithm

- initialize $\theta$ (usually to the zero vector)
- repeat:
  - Pick a training example $(x, y)$
  - Model predicts $\hat{y}$ that maximizes score($x, \hat{y}$)
  - Update weights by a step of size $\varepsilon > 0$:
    \[
    \theta = \theta + \varepsilon \cdot (f(x, y) - f(x, \hat{y}))
    \]
    call this $\Delta$

If model prediction was wrong ($y \neq \hat{y}$), then we must have score($x, y$) $\leq$ score($x, \hat{y}$) instead of $>$ as we want.
Equivalently, $\theta \cdot f(x, y) \leq \theta \cdot f(x, \hat{y})$ but we want it $>$
Equivalently, $\theta \cdot (f(x, y) - f(x, \hat{y})) = \theta \cdot \Delta \leq 0$ but we want it $> 0$.
So update increases $\theta \cdot \Delta$, to $(\theta + \varepsilon \cdot \Delta) \cdot \Delta = \theta \cdot \Delta + \varepsilon \cdot ||\Delta||^2 \geq 0$
\[ p(\text{parse} \mid \text{sentence}) \quad \text{score}(\text{sentence}, \text{parse}) \]

\[
\begin{array}{c|c|c}
\text{Time flies like an arrow} & \text{VP} & \text{PP} \\
\text{S} & \text{NP} & \text{N} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{Time flies like an arrow} & \text{VP} & \text{PP} \\
\text{S} & \text{NP} & \text{N} \\
\end{array}
\]
Finding the best $y$ given $x$

- At **both** training & test time, given input $x$, perceptron picks $y$ that maximizes $\text{score}(x,y)$

$$\text{score}(x,y) = \sum_k \theta_k f_k(x,y)$$

- How do we compute that crucial prediction??
  - Easy when only a few candidates $y$ (text classification, WSD, ...)
    - Just try each $y$ in turn.
  - Harder for structured prediction: **but you now know how!**
    - Find the best string, path, or tree ...
    - That’s what Viterbi-style or Dijkstra-style algorithms are for.
      - That is, use dynamic programming to find the score of the best $y$.
      - Then follow backpointers to recover the $y$ that achieves that score.
An Alternative Tradition

- Old AI hacking technique:
  - Possible parses (or whatever) have scores.
  - Pick the one with the best score.
  - How do you define the score?
    - Completely ad hoc!
    - Throw anything you want into the stew
      - Add a bonus for this, a penalty for that, etc.
    - "Learns" over time
      - as you adjust bonuses and penalties by hand to improve performance.
    - Total kludge, but totally flexible too …

Exposé at 9

Probabilistic Revolution
Not Really a Revolution, Critics Say

Log-probabilities no more than scores in disguise

“We’re just adding stuff up like the old corrupt regime did,”
admits spokesperson

really so alternative?
Nuthin’ but adding weights

- **n-grams:** \[ \ldots + \log p(w_7 \mid w_5, w_6) + \log p(w_8 \mid w_6, w_7) + \ldots \]
- **PCFG:** \[ \log p(NP \ VP \mid S) + \log p(Papa \mid NP) + \log p(VP \ PP \mid VP) \ldots \]
- **HMM tagging:** \[ \ldots + \log p(t_7 \mid t_5, t_6) + \log p(w_7 \mid t_7) + \ldots \]
- **Noisy channel:** \[ [ \log p(\text{source}) ] + [ \log p(\text{data} \mid \text{source}) ] \]
- **Cascade of composed FSTs:**
  \[ [ \log p(A) ] + [ \log p(B \mid A) ] + [ \log p(C \mid B) ] + \ldots \]
- **Naïve Bayes:**
  \[ \log p(\text{Class}) + \log p(\text{feature1} \mid \text{Class}) + \log p(\text{feature2} \mid \text{Class}) \ldots \]
- **Note:** Here we’re using \(+\log\text{prob}\) not \(–\log\text{prob}\):
  i.e., bigger weights are **better**.
Nuthin’ but adding weights

- **n-grams:** \( \ldots + \log p(w_7 \mid w_5, w_6) + \log p(w_8 \mid w_6, w_7) + \ldots \)

- **PCFG:** \( \log p(NP \ VP \mid S) + \log p(Papa \mid NP) + \log p(VP \ PP \mid VP) \ldots \)
  - Score of a parse is its total weight
  - The weights we add up have always been log-probs (\( \leq 0 \))
  - but what if we changed that?

- **HMM tagging:** \( \ldots + \log p(t_7 \mid t_5, t_6) + \log p(w_7 \mid t_7) + \ldots \)

- **Noisy channel:** \([ \log p(\text{source}) ] + [ \log p(\text{data} \mid \text{source}) ]\)

- **Cascade of FSTs:**
  \[
  [ \log p(A) ] + [ \log p(B \mid A) ] + [ \log p(C \mid B) ] + \ldots
  \]

- **Naïve Bayes:**
  \[
  \log(\text{Class}) + \log(\text{feature1} \mid \text{Class}) + \log(\text{feature2} \mid \text{Class}) + \ldots
  \]
What if our weights were arbitrary real numbers?
Change $\log p(\text{this} \mid \text{that})$ to $\theta(\text{this} ; \text{that})$

- **n-grams:** $\ldots + \log p(w_7 \mid w_5, w_6) + \log p(w_8 \mid w_6, w_7) + \ldots$
- **PCFG:** $\log p(\text{NP VP} \mid \text{S}) + \log p(\text{Papa} \mid \text{NP}) + \log p(\text{VP PP} \mid \text{VP}) \ldots$
- **HMM tagging:** $\ldots + \log p(t_7 \mid t_5, t_6) + \log p(w_7 \mid t_7) + \ldots$
- **Noisy channel:** $[\log p(\text{source})] + [\log p(\text{data} \mid \text{source})]$
- **Cascade of FSTs:**
  $[\log p(A)] + [\log p(B \mid A)] + [\log p(C \mid B)] + \ldots$
- **Naïve Bayes:**
  $\log p(\text{Class}) + \log p(\text{feature1} \mid \text{Class}) + \log p(\text{feature2} \mid \text{Class}) \ldots$
What if our weights were arbitrary real numbers?
Change \( \log p(\text{this} \mid \text{that}) \) to \( \theta(\text{this} ; \text{that}) \)

- **n-grams:** \( ... + \theta(w7 ; w5, w6) + \theta(w8 ; w6, w7) + ... \)
- **PCFG:** \( \theta(\text{NP VP} ; \text{S}) + \theta(\text{Papa} ; \text{NP}) + \theta(\text{VP PP} ; \text{VP}) \) ...
- **HMM tagging:** \( ... + \theta(t7 ; t5, t6) + \theta(w7 ; t7) + ... \)
- **Noisy channel:** \( [ \theta(\text{source}) ] + [ \theta(\text{data} ; \text{source}) ] \)
- **Cascade of FSTs:**
  \[
  \left[ \theta(\text{A}) \right] + \left[ \theta(\text{B} ; \text{A}) \right] + \left[ \theta(\text{C} ; \text{B}) \right] + ...
  \]
- **Naïve Bayes:**
  \( \theta(\text{Class}) + \theta(\text{feature1} ; \text{Class}) + \theta(\text{feature2} ; \text{Class}) \) ...

In practice, \( \theta \) is a hash table
Maps from feature name (a string or object) to feature weight (a float)
e.g., \( \theta(\text{NP VP} ; \text{S}) \) = weight of the \( S \rightarrow NP \ VP \) rule, say -0.1 or +1.3
What if our weights were arbitrary real numbers?

Change \( \log p(\text{this} \mid \text{that}) \) to \( \theta(\text{this} ; \text{that}) \theta(\text{that} & \text{this}) \) [prettier name]

- **n-grams**: \( \ldots + \theta(w_5 \ w_6 \ w_7) + \theta(w_6 \ w_7 \ w_8) + \ldots \)

- **PCFG**: \( \theta(S \rightarrow \text{NP VP}) + \theta(\text{NP} \rightarrow \text{Papa}) + \theta(\text{VP} \rightarrow \text{VP PP}) \ldots \)

- **HMM tagging**: \( \ldots + \theta(t_5 \ t_6 \ t_7) + \theta(t_7 \rightarrow w_7) + \ldots \)

- **Noisy channel**: \( [ \theta(\text{source}) ] + [ \theta(\text{source}, \text{data}) ] \)

- **Cascade of FSTs**: \( [ \theta(A) ] + [ \theta(A, B) ] + [ \theta(B, C) ] + \ldots \)

- **Naïve Bayes**: (multi-class) logistic regression
  \( \theta(\text{Class}) + \theta(\text{Class}, \text{feature 1}) + \theta(\text{Class}, \text{feature 2}) \ldots \)

In practice, \( \theta \) is a hash table
Maps from feature name (a string or object) to feature weight (a float)
e.g., \( \theta(S \rightarrow \text{NP VP}) = \) weight of the \( S \rightarrow \text{NP VP} \) rule, say -0.1 or +1.3
What if our weights were arbitrary real numbers?
Change \( \log p(\text{this} | \text{that}) \) to \( \theta(\text{that} \& \text{this}) \)

- **n-grams:** \( \ldots + \theta(w5 \ w6 \ w7) + \theta(w6 \ w7 \ w8) + \ldots \)
  - Best string is the one whose trigrams have the highest total weight

- **PCFG:** \( \theta(S \rightarrow NP \ VP) + \theta(NP \rightarrow \text{Papa}) + \theta(VP \rightarrow VP \ PP) + \ldots \)
  - Best parse is one whose rules have highest total weight

- **HMM tagging:** \( \ldots + \theta(t5 \ t6 \ t7) + \theta(t7 \rightarrow w7) + \ldots \)
  - Best tagging has highest total weight of all transitions and emissions

- **Noisy channel:** \([ \theta(\text{source}) ] + [ \theta(\text{source, data}) ] \)
  - To guess source: max (weight of source + weight of source-data match)

- **Naïve Bayes:** \( \theta(\text{Class}) + \theta(\text{Class, feature 1}) + \theta(\text{Class, feature 2}) \)
  - Best class maximizes prior weight + weight of compatibility with features (multi-class) logistic regression
What if our weights were arbitrary real numbers?
Change \( \log p(\text{this} | \text{that}) \) to \( \theta(\text{that} \& \text{this}) \)

**All our algorithms still work!**

We’ll just add up arbitrary feature weights \( \theta \) that might not be log conditional probabilities

(they might even be positive!)

Total score(\(x, y\)) can’t be interpreted anymore as \( \log p(x, y) \)

But we can still find the highest-scoring \(y\)

(\text{using a Viterbi algorithm})
Given sentence $x$
You know how to find max-score parse $y$ (or min-cost parse as shown)
- Provided that the score of a parse = total score of its rules

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NP</strong> 3</td>
<td><strong>NP</strong> 10</td>
<td><strong>NP</strong> 24</td>
<td><strong>NP</strong> 4</td>
<td><strong>NP</strong> 18</td>
</tr>
<tr>
<td>Vst 3</td>
<td>S 8</td>
<td>S 22</td>
<td>S 21</td>
<td>S 22</td>
</tr>
<tr>
<td>S 13</td>
<td></td>
<td>S 27</td>
<td>PP 12</td>
<td>VP 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NP 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NP 10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td><strong>VP</strong></td>
<td><strong>PP</strong></td>
<td><strong>NP</strong></td>
<td><strong>Det</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Time flies like an arrow

**Production Rules:**
1. $S \rightarrow NP \ VP$
2. $S \rightarrow S \ PP$
3. $NP \rightarrow NP \ VP$
4. $NP \rightarrow NP \ PP$
5. $VP \rightarrow V \ NP$
6. $S \rightarrow Vst \ NP$
7. $VP \rightarrow VP \ PP$
8. $NP \rightarrow Det \ N$
9. $VP \rightarrow VP \ PP$
10. $PP \rightarrow P \ NP$

- **Score Breakdown:**
  - $time$ 1: 3
  - $flies$ 2: 10
  - $like$ 3: 8
  - $an$ 4: 13
  - $arrow$ 5: 24
  - $Det$ 1: 10
  - $N$ 8
Given word sequence x
You know how to find max-score tag sequence y

- Provided that the score of a tagged sentence
  = total score of its emissions and transitions
- These don’t have to be log-probabilities!
  - Emission scores assess tag-word compatibility
  - Transition scores assess goodness of tag bigrams

Bill directed a cortege of autos through the dunes
Given upper string $x$
You know how to find max-score path that accepts $x$ (or min-cost path)
- Provided that the score of a path = total score of its arcs
  - Then the best lower string $y$ is the one along that best path
    - (So in effect, $\text{score}(x,y)$ is score of best path that transduces $x$ to $y$)
  - Q: How do you make sure that the path accepts $x$, such as $\text{aaaaaba}$?
    - A: Compose with straight-line automaton for $x$, then find best path.
Running Example: Predict a Tagging

Given word sequence \( x \)
Find max-score tag sequence \( y \)

\[
\text{score}(\begin{array}{cccc} \text{BOS} & \text{N} & \text{V} & \text{EOS} \end{array}) = ?
\]

Time flies

\[
\text{score}(x, y) = \sum_k \theta_k f_k(x, y)
\]

So what are the features?
Let’s start with the usual emission and transition features ...
Running Example: Predict a Tagging

Given word sequence \( x \)
Find max-score tag sequence \( y \)

\[
\text{score}(BOS, N, V, EOS) = \sum_k \theta_k f_k(x, y)
\]

\[
\begin{align*}
\text{Time flies} & \\
& = \theta_{BOS,N} + \theta_{N,\text{Time}} + \theta_{N,V} + \theta_{V,\text{flies}} + \theta_{V,EOS}
\end{align*}
\]

So what are the features?
Let’s start with the usual emission and transition features ...
Running Example: Predict a Tagging

Given word sequence \( x \)
Find max-score tag sequence \( y \)

\[
\text{score}( BOS \quad N \quad V \quad EOS ) = \sum_k \theta_k f_k(x,y)
\]

\[
= \theta_{BOS,N} + \theta_{N,\text{Time}} + \theta_{N,V} + \theta_{V,\text{flies}} + \theta_{V,\text{EOS}}
\]

score includes \( f_{N,V}(x,y) \) copies of this feature weight = one copy per N V token

For each \( t \in \text{Tags}, \ w \in \text{Words} \):
define \( f_{t,w}(x,y) = \) count of emission \( t \ w \)
\[
= | \{ i : 1 \leq i \leq |x|, \ y_i = t, \ x_i = w \} |
\]

For each \( t, \ t' \in \text{Tags} \):
define \( f_{t,t'}(x,y) = \) count of transition \( t \ t' \)
\[
= | \{ i : 0 \leq i \leq |x|, \ y_i = t, \ y_{i+1} = t' \} |
\]
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths) Viterbi algorithm can find the highest-scoring tagging
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths) Viterbi algorithm can find the highest-scoring tagging
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths) Viterbi algorithm can find the highest-scoring tagging
Set arc weights so that path weight = tagging score
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths) Viterbi algorithm can find the highest-scoring tagging Set arc weights so that path weight = tagging score
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths) Viterbi algorithm can find the highest-scoring tagging Set arc weights so that path weight = tagging score

In structured perceptron, weights \( \theta \) are no longer log-probs. They’re tuned by structured perceptron to make the correct path outscore others.
Why would we switch from probabilities to scores?

1. “Discriminative” training (e.g., perceptron) might work better.
   - It tries to optimize weights to actually predict the right $y$ for each $x$.
   - More important than maximizing $\log p(x, y) = \log p(y|x) + \log p(x)$, as we’ve been doing in HMMs and PCFGs.
   - Satisfied once the right $y$ wins. The example puts no more pressure on the weights to raise $\log p(y|x)$. And never pressures us to raise $\log p(x)$.

2. Having more freedom in the weights might help?
   - Now weights can be positive or negative.
   - Exponentiated weights no longer have to sum to 1.
   - But turns out new $\theta$ vectors can’t do more than the old restricted ones.
     - Roughly, for every WCFG there’s an equivalent PCFG.
     - Though it’s true a regularizer might favor one of the new ones.

3. We can throw lots more features into the stewpot.
   - Allows model to capture more of the useful predictive patterns!
   - So, what features can we throw in efficiently?
When can you efficiently choose best $y$?

- “Provided that the score of a tagged sentence = total score of its transitions and emissions”
- “Provided that the score of a path = total score of its arcs”
- “Provided that the score of a parse = total score of its rules”

This implies certain kinds of features in linear model ...

$$\text{score}(x, y) = \sum_k \theta_k f_k(x, y)$$

e.g, $\theta_3 =$ score of N V tag bigram

$f_3(x,y) =$ # times N V appears in $y$
When can you efficiently choose best $y$?

- “Provided that the score of a tagged sentence = total score of its transitions and emissions”
- “Provided that the score of a path = total score of its arcs”
- “Provided that the score of a parse = total score of its rules”

This implies certain kinds of features in linear model ...

\[
\text{score}(x, y) = \sum_{k} \theta_k f_k(x, y)
\]

- e.g, $\theta_3 = \text{score of VP } \rightarrow \text{ VP PP}$
- $f_3(x, y) = \# \text{ times VP } \rightarrow \text{ VP PP appears in } y$
When can you efficiently choose best $y$?

- “Provided that the score of a tagged sentence
  $= \text{total score of its transitions and emissions}”$
- “Provided that the score of a path $= \text{total score of its arcs}”$
- “Provided that the score of a parse $= \text{total score of its rules}”$

This implies certain kinds of features in linear model ...

$$\text{score}(x, y) = \sum_{k} \theta_k f_k(x, y)$$

More generally: make a list of interesting substructures. The feature $f_k(x, y)$ counts tokens of $k$th substructure in $(x, y)$. So far, the substructures $= \text{transitions, emissions, arcs, rules}$. But model \textbf{could} use any features ... what ones are \textbf{efficient}?
1. Single-rule substructures

- Count of \( VP \rightarrow VP \ PP \)

*Time flies like an arrow*
1. Single-rule substructures

- Count of $\text{VP} \rightarrow \text{VP PP}$ (looks at $y$ only)
- Count of $\text{V} \rightarrow \text{flies}$ (looks at both $x$ and $y$)
2. Within-rule substructures

- Count of VP with a PP child

Time flies like an arrow
2. Within-rule substructures

- Count of VP with a PP child
- Count of any node with a PP right child

Time flies like an arrow
2. Within-rule substructures

- Count of VP with a PP child
- Count of any node with a PP right child
- Count of any node with a PP right child and whose label matches left child’s label

Time flies like an arrow
2. Within-rule substructures

Efficient?
Yes: the weight that CKY uses for VP → VP PP is the total weight of all of its within-rule features.

Some of these features fire on both VP → VP PP and NP → NP PP. So they're really backoff features.

- Count of VP with a PP child
- Count of any node with a PP right child
- Count of any node with a PP right child and whose nonterm matches left child’s nonterm

Time flies like an arrow
3. Cross-rule substructures

- Count of “flies” as a verb with subject “time”
3. Cross-rule substructures

- Count of “flies” as a verb with subject “time”
- Count of NP $\rightarrow$ D N when the NP is the object of a preposition
3. Cross-rule substructures

- Count of “flies” as a verb with subject “time”
- Count of NP → D N when the NP is the object of a preposition
- Count of VP constituents that contain a V

Two such VPs, so feature fires twice on this (x, y) pair
3. Cross-rule substructures

- Count of “flies” as a verb with subject “time”
- Count of NP → D N when the NP is the object of a preposition
- Count of VPs that contain a V

Efficient? Sort of. For CKY to work, must add attributes to the nonterminals so that these features can now be detected within-rule. That enlarges the grammar.

What’s the analogue in FSMs? Splitting states to remember more history.
4. Global features

- Count of “NP and NP” when the two NPs have very different size or structure [this feature has weight < 0]
- The number of PPs is even
- The depth of the tree is prime 😊
- Count of the tag bigram V P in the preterminal seq
4. Global features

Efficient? Depends on whether you can do it with attributes. If you have infinitely many nonterminals, it’s not technically a PCFG anymore, but CKY might still apply.

Or stop relying only on dynamic programming. Start using approximate or exact general methods for combinatorial optimization. Hot area!

- Count of “NP and NP” when the two NPs have very different size or structure [this feature has weight < 0]
- The number of PP s is even
- The depth of the tree is prime 😊
- Count of the tag bigram V P in the preterminal seq

Time flies like an arrow
5. Context-specific features

Take any efficient feature that counts a substructure. Modify it to count only tokens appearing in a particular red context.

Time flies like an arrow

---

Take any efficient feature that counts a substructure. Modify it to count only tokens appearing in a particular red context.
5. Context-specific features

Take any efficient feature that counts a substructure. Modify it to count only tokens appearing in a particular red context.

Count of $VP \rightarrow VP \ PP$ whose first word is “flies”
Take any efficient feature that counts a substructure. Modify it to count only tokens appearing in a particular red context.

- Count of $VP \rightarrow VP \ PP$ whose first word is “flies”
- Count of $VP \rightarrow VP \ PP$ whose right child has width 3
5. Context-specific features

Take any efficient feature that counts a substructure. Modify it to count only tokens appearing in a particular red context.

- Count of $VP \rightarrow VP \ PP$ whose first word is “flies”
- Count of $VP \rightarrow VP \ PP$ whose right child has width 3
- Count of $VP \rightarrow VP \ PP$ at the end of the input
5. Context-specific features

Take any efficient feature that counts a substructure. Modify it to count only tokens appearing in a particular red context.

- Count of $\text{VP} \rightarrow \text{VP PP}$ whose first word is "flies"
- Count of $\text{VP} \rightarrow \text{VP PP}$ whose right child has width 3
- Count of $\text{VP} \rightarrow \text{VP PP}$ at the end of the input
- Count of $\text{VP} \rightarrow \text{VP PP}$ right after a capitalized word

Time flies like an arrow
5. Context-specific features

Take any efficient feature that counts a substructure. Modify it to count only tokens appearing in a particular red context.

Still efficient? Amazingly, yes!

Features like these have played a big role in improving real-world accuracy of NLP systems.

Time flies like an arrow

- Count of $VP \rightarrow VP \ PP$ whose first word is “flies”
- Count of $VP \rightarrow VP \ PP$ whose right child has width 3
- Count of $VP \rightarrow VP \ PP$ at the end of the input
- Count of $VP \rightarrow VP \ PP$ right after a capitalized word
5. Context-specific features

When CKY combines [1,2] with [2,5] using the rule $\text{VP} \rightarrow \text{VP PP}$, it is using that rule in a particular context.

The weight of the rule in that context can sum over features that look at the context (i.e., the red information). Doesn’t change CKY runtime!
**Same approach for tagging ...**

- Previous slides used *parsing* as an example.
  - Given a *sentence* of length \( n \), reconstructing the best *tree* takes time \( O(n^3) \).
    - Specifically, \( O(Gn^3) \) where \( G = \# \) of grammar rules.

- As we’ll see, many NLP tasks only need to *tag* the words (not necessarily with parts of speech).
  - Don’t need training trees, only training tags.
  - Reconstructing the best *tagging* takes only time \( O(n) \).
    - Specifically, \( O(Gn) \) where \( G = \# \) of legal tag bigrams.
  - It’s just the Viterbi tagging algorithm again.
    - But now score is a sum of many feature weights ...
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths)
Set arc weights so that path weight = tagging score

Let’s add lots of other weights to the arc score!
Does V:flies score highly? Depends on features of N V and V flies in context (at word 2 of sentence x).
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths)
Set arc weights so that path weight = tagging score

\[
\theta_{\text{BOS}, N} + \theta_{N, \text{Time}} + \theta_{N, V} + \theta_{V, \text{flies}} + \theta_{V, \text{flies}, i=2} + \theta_{\text{capitalized }, N, V} + \theta_{V} + ... \]

\[
\theta_{V, \text{EOS}}
\]
Context-specific tagging features

Time flies like an arrow

- Count of tag P as the tag for “like”

In an HMM, the weight of this feature would be the log of an emission probability

But in general, it doesn’t have to be a log probability
Context-specific tagging features

Time flies like an arrow

- Count of tag P as the tag for “like”
- Count of tag P
Context-specific tagging features

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
Context-specific tagging features

Time flies like an arrow

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P

In an HMM, the weight of this feature would be the log of an emission probability

But in general, it doesn’t have to be a log probability
Context-specific tagging features

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P
- Count of tag bigram V P followed by “an”
Context-specific tagging features

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P
- Count of tag bigram V P followed by “an”
- Count of tag bigram V P where P is the tag for “like”
Context-specific tagging features

Time flies like an arrow

- Count of tag P as the tag for “like”
- Count of tag P
- Count of tag P in the middle third of the sentence
- Count of tag bigram V P
- Count of tag bigram V P followed by “an”
- Count of tag bigram V P where P is the tag for “like”
- Count of tag bigram V P where both words are lowercase
More expensive tagging features

Count of tag trigram N V P?
- A bigram tagger can only consider within-bigram features: only look at 2 adjacent blue tags (plus arbitrary red context).
- So here we need a trigram tagger, which is slower.
- As an FST, its state would remember two previous tags.

We take this arc once per N V P triple, so its weight is the total weight of the features that fire on that triple.
More expensive tagging features

- Count of tag trigram N V P?
  - A bigram tagger can only consider within-bigram features: only look at 2 adjacent blue tags (plus arbitrary red context).
  - So here we need a trigram tagger, which is slower.

- Count of “post-verbal” nouns? (“discontinuous bigram” V N)
  - An n-gram tagger can only look at a narrow window.
  - So here we need an FSM whose states remember whether there was a verb in the left context.
How might you come up with the features that you will use to score \((x, y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x, y)\).

For position \(i\) in a tagging, these might include:

- Full name of tag \(i\)
- First letter of tag \(i\) (will be "N" for both "NN" and "NNS")
- Full name of tag \(i-1\) (possibly BOS); similarly tag \(i+1\) (possibly EOS)
- Full name of word \(i\)
- Last 2 chars of word \(i\) (will be "ed" for most past-tense verbs)
- First 4 chars of word \(i\) (why would this help?)
- "Shape" of word \(i\) (lowercase/capitalized/all caps/numeric/…)
- Whether word \(i\) is part of a known city name listed in a "gazetteer"
- Whether word \(i\) appears in thesaurus entry \(e\) (one attribute per \(e\))
- Whether \(i\) is in the middle third of the sentence
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes (“basic features”) that you can compute at each position in \((x,y)\).

For a node \(n\) in a parse tree that covers the substring \((i,j)\):

- Nonterminal at \(n\)
- Nonterminal at first child of \(n\), or “null” if child is a word
- Nonterminal at second child of \(n\), or “null” if only one child
- Constituent width \(j-i\)
- Whether \(j-i \leq 3\) (true/false)
- Whether \(j-i \leq 10\) (true/false)
- Words \(i+1\) and \(j\) (first and last words of constituent)
- Words \(i\) and \(j+1\) (words immediately before and after constituent)
- Suffixes, prefixes, shapes, and categories of all of these words
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x,y)\).
2. Now conjoin them into various “feature templates.”

E.g., template 7 might be \((\text{tag}(i), \text{tag}(i+1), \text{suffix2}(i+2))\).

At each position of \((x,y)\), exactly one of the many template7 features will fire:

At \(i=0\), we see an instance of “template7\(\equiv\)\((\text{BOS},\text{N},-\text{es})\)”, so we add one copy of that feature’s weight to \(\text{score}(x,y)\)
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x,y)\).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be \((\text{tag}(i), \text{tag}(i+1), \text{suffix2}(i+2))\).

At each position of \((x,y)\), exactly one of the many template7 features will fire:

```
N V P D N
```

**Time flies like an arrow**

At \(i=1\), we see an instance of "template7\(^7\)\!=(\text{N},\text{V},-\text{ke})" so we add one copy of that feature’s weight to \(\text{score}(x,y)\).
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes (“basic features”) that you can compute at each position in \((x,y)\).
2. Now conjoin them into various “feature templates.”

E.g., template 7 might be \((\text{tag}(i), \text{tag}(i+1), \text{suffix2}(i+2))\).

At each position of \((x,y)\), exactly one of the many template7 features will fire:

\[
\begin{array}{cccccc}
N & V & P & D & N \\
\text{Time flies like an arrow}
\end{array}
\]

At \(i=2\), we see an instance of “template7=\((N,V,-an)\)” so we add one copy of that feature’s weight to \(\text{score}(x,y)\).
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x,y)\).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be \((\text{tag}(i), \text{tag}(i+1), \text{suffix}2(i+2))\).

At each position of \((x,y)\), exactly one of the many template7 features will fire:

![Example sentence: Time flies like an arrow](image)

At \(i=3\), we see an instance of "template7=(P,D,-ow)" so we add one copy of that feature’s weight to \(\text{score}(x,y)\)
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x,y)\).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be \((\text{tag}(i), \text{tag}(i+1), \text{suffix}2(i+2))\).

At each position of \((x,y)\), exactly one of the many template7 features will fire:

```
N   V   P   D   N
```

*Time flies like an arrow*

At \(i=4\), we see an instance of "template7=(D,N,-)" so we add one copy of that feature’s weight to \(\text{score}(x,y)\)
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x,y)\).
2. Now conjoin them into various "feature templates."

E.g., template 7 might be \((\text{tag}(i), \text{tag}(i+1), \text{suffix2}(i+2))\). This template gives rise to many features, e.g.:

\[
\text{score}(x,y) = \ldots \\
+ \theta[\text{"template7=}(P,D,-ow)\text{"]} \ast \text{count("template7=} (P,D,-ow)\text{")} \\
+ \theta[\text{"template7=} (D,D,-xx)\text{"]} \ast \text{count("template7=} (D,D,-xx)\text{")} \\
+ \ldots
\]

With a handful of feature templates and a large vocabulary, you can easily end up with millions of features.
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x,y)\).
2. Now conjoin them into various “feature templates.”

E.g., template 7 might be \((\text{tag}(i), \text{tag}(i+1), \text{suffix2}(i+2))\).

Note: Every template should mention at least some blue.

- Given an input \(x\), a feature that only looks at red will contribute the same weight to \(\text{score}(x,y_1)\) and \(\text{score}(x,y_2)\).
- So it can’t help you choose between outputs \(y_1, y_2\).
How might you come up with the features that you will use to score \((x,y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x,y)\).
2. Now conjoin them into various "feature templates."
3. Train your system!

   - What if you had too many features?
     - That’s what regularization is for. Prevents overfitting.
     - An L1 regularizer will do “feature selection” for you. Keeps a feature’s weight at 0 if it didn’t help enough on the training data.
     - Fancier extensions of L1 will even do feature template selection.
       - If training throws out a template, you get a test-time speedup.
       - (Ordinarily at test time, at every position, you’d have to construct a feature from that template & look up its weight in a hash table.)

Group lasso, graphical lasso, feature induction in random fields, meta-features …
How might you come up with the features that you will use to score \((x, y)\)?

1. Think of some attributes ("basic features") that you can compute at each position in \((x, y)\).
2. Now **conjoin** them into various "feature templates."
3. Train your system!
   - What if you had too many features?
   - What if you didn’t have enough features?
     - Then your system will have some errors.
     - Study errors and come up with features that might help fix them.
     - Maybe try to learn features automatically (e.g., "deep learning").
     - Alternatively, the "kernel trick" lets you expand to mind-bogglingly big (even infinite) feature sets. E.g., all 5-way conjunctions of existing features, including conjunctions that don’t stay within an n-gram!
   - Runtime no longer scales up with the # of features that fire on a sentence. But now it scales up with the # of training examples.

**Check out** "kernelized perceptron."

**But the trick started with kernel SVMs.**
Probabilists Rally Behind Paradigm

“.2, .4, .6, .8! We’re not gonna take your bait!”

1. Maybe we like our training criterion better than perceptron
   - Modeling the true probability distribution may generalize better
2. Our model offers a whole distribution, not just one output:
   - How sure are we that \( y \) is the correct parse? (confidence)
   - What’s the expected error of parse \( y \)? (Bayes risk)
   - What parse \( y \) has minimum expected error? (posterior decoding)
   - Marginal prob that [time flies] is NP? (soft feature for another system)
3. Our results can be meaningfully combined ⇒ modularity!
   - Train several systems and multiply their conditional probabilities
   - \( p(\text{English text}) \times p(\text{English phonemes} \mid \text{English text}) \times p(\text{Jap. phonemes} \mid \text{English phonemes}) \times p(\text{Jap. text} \mid \text{Jap. phonemes}) \times p(\text{semantics}) \times p(\text{syntax} \mid \text{semantics}) \times p(\text{morphology} \mid \text{syntax}) \times p(\text{phonology} \mid \text{morphology}) \times p(\text{sounds} \mid \text{phonology}) \)
Probabilists Regret Being Bound by Principle

1. Those context-specific features sure seem helpful!
2. And even with context-free features, discriminative training generally gets better accuracy.

- Fortunately, both of these deficiencies can be fixed within a **probabilistic** framework. 😊
  - Perceptron only learns how to score structures. The scores may use rich features, but don’t have a probabilistic interpretation.
  - Let’s keep the same scoring functions (linear functions on the same features). But now interpret $\text{score}(x, y)$ as $\log p(y | x)$.
  - As usual for such log-linear models, train the weights so that the events in training data have high conditional probability $p(y | x)$. Slightly different from perceptron training. But like perceptron, we’re training the weights to discriminate among $y$ values, rather than to predict $x$. 
Time flies like an arrow
Generative processes

1. Those context-specific features sure seem helpful!
2. Even with the same features, discriminative training generally gets better accuracy.

- Fortunately, both of these deficiencies can be fixed within a **probabilistic** framework. 😊
  - Our PCFG, HMM, and probabilistic FST frameworks relied on modeling the probabilities of individual context-free moves:
    - \( p(\text{rule} \mid \text{nonterminal}) \), \( p(\text{word} \mid \text{tag}) \), \( p(\text{tag} \mid \text{previous tag}) \), \( p(\text{transition} \mid \text{state}) \)
    - Perhaps each of these was a log-linear conditional probability.
    - Our models multiplied them all to get a joint probability \( p(x,y) \).
    - Instead, let’s model \( p(y \mid x) \) as a single log-linear distribution …
1. Those context-specific features sure seem helpful!
2. Even with the same features, discriminative training generally gets better accuracy.

- Fortunately, both of these deficiencies can be fixed within a **probabilistic** framework. ☺

<table>
<thead>
<tr>
<th>Markov Random Field (MRF)</th>
<th>Conditional Random Field (CRF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(x, y) = \frac{1}{Z} \exp \theta f(x, y) )</td>
<td>( p(y</td>
</tr>
<tr>
<td>Train to maximize ( \log p(x, y) )</td>
<td>Train to maximize ( \log p(y</td>
</tr>
<tr>
<td>Generates ( x, y ) “all at once.”</td>
<td>Generates ( y ) “all at once” given ( x ).</td>
</tr>
<tr>
<td>Scores result as a whole, not individual generative steps.</td>
<td>Discriminative like perceptron ... and efficient for same features.</td>
</tr>
</tbody>
</table>
Finding the best y given x

\[ \text{score}(x, y) = \sum_{k} \theta_k f_k(x, y) \]

- How do you make predictions given input \( x \)?
- Can just use the same Viterbi algorithms again!
- **Perceptron** picks \( y \) that maximizes \( \text{score}(x, y) \).
- **CRF** defines \( p(y \mid x) = (1/Z(x)) \exp \text{score}(x, y) \).
  - For a single output, could pick \( y \) that maximizes \( p(y \mid x) \).
    - This “1-best” prediction is the single \( y \) that is most likely to be completely right (according to your trained model).
  - But that’s exactly the \( y \) that maximizes \( \text{score}(x, y) \).
    - Why? \( \exp \) is an increasing function, and \( 1/Z(x) \) is constant.
- The only difference is in how \( \theta \) is trained.
Perceptron Training Algorithm

- initialize $\theta$ (usually to the zero vector)
- repeat:
  - Pick a training example $(x, y)$
  - Current $\theta$ predicts $\hat{y}$ maximizing score($x, \hat{y}$)
  - Update weights by a step of size $\epsilon > 0$:
    $$\theta = \theta + \epsilon \cdot (f(x,y) - f(x,\hat{y}))$$
initialize $\theta$ (usually to the zero vector)

repeat:

- Pick a training example $(x,y)$
- Current $\theta$ predicts $\hat{y}$ maximizing score $(x,\hat{y})$
- Update weights by a step of size $\varepsilon > 0$:
  $$\theta = \theta + \varepsilon \cdot (f(x,y) - f(x,\hat{y}))$$

defines a distribution $p(\hat{y} | x)$

expected features of a random $\hat{y}$ chosen from the distribution:
$$\sum_{\hat{y}} p(\hat{y} | x) f(x,\hat{y})$$
CRF Training Algorithm

- initialize $\theta$ (usually to the zero vector)
- repeat:
  - Pick a training example $(x,y)$
  - Current $\theta$ defines a distribution $p(\hat{y} | x)$
  - Update weights by a step of size $\epsilon > 0$:
    $$
    \theta = \theta + \epsilon \cdot (f(x,y) - \sum_{\hat{y}} p(\hat{y} | x) f(x,\hat{y}))
    $$
  - Update $\epsilon$.

That is, we’re training a conditional log-linear model $p(y | x)$ by stochastic gradient ascent as usual.

(Should add a regularizer, hence a step to update weights toward 0.)

But now $y$ is a **big structure** like trees or taggings.

Must get smaller but not too fast; can use $\epsilon = 1/(t+1000)$ on iter $t$. 
CRF Training Algorithm

- initialize \( \theta \) (usually to the zero vector)
- repeat:
  - Pick a training example \((x,y)\)
  - Current \( \theta \) defines a distribution \( p(\hat{y} \mid x) \)
  - Update weights by a step of size \( \epsilon > 0 \):
    \[
    \theta = \theta + \epsilon \cdot (f(x,y) - \sum_{\hat{y}} p(\hat{y} \mid x) f(x,\hat{y}))
    \]
    observed - expected features

How do we compute the expected features?
Forward-backward or inside-algorithm tells us expected count of each substructure (transition / emission / arc / rule). So we can iterate over that substructure’s features.
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths)

\[ p(\text{path} \mid x) = \frac{1}{Z(x)} \exp(\text{sum of } \theta \text{ values on path}) \]

\[ \theta_{\text{BOS},N} + \theta_{N,\text{Time}} \]

\[ \theta_{N,V} + \theta_{V,\text{flies}} \]

\[ \theta_{V,\text{EOS}} \]

perceptron: \((\oplus, \otimes) = (\text{max}, +)\)

CRF: \((\oplus, \otimes) = (\log+, +)\)

treat scores as log-probs

Run forward algorithm in this semiring to get

\[ \log Z(x) = \text{“total” (log+) weight of all paths} \]
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths)

\[ p(path \mid x) = \frac{1}{Z(x)} \exp(\text{sum of } \theta \text{ values on } path) = \frac{1}{Z(x)} (\text{product of } \phi \text{ values on } path) \]

where we define each \( \phi_k = \exp \theta_k \)

perceptron: \((\oplus, \otimes) = (\max, +)\)

CRF: \((\oplus, \otimes) = (\log+, +)\) \((+,*\) \)

if you prefer simpler semiring ...

forward algorithm in this semiring to get

\[ \log Z(x) = \text{total weight of all paths} \]
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths)
\[ p(path | x) = \frac{1}{Z(x)} \exp(\text{sum of } \theta \text{ values on } path) \]
\[ = \frac{1}{Z(x)} \text{(product of } \phi \text{ values on } path) \]
where we define each \( \phi_k = \exp \theta_k \)

perceptron: \((\oplus, \otimes) = (\max, +)\)

CRF: \((\oplus, \otimes) = (\log+, +)\) \((+, \ast)\)

forward-backward algorithm to get expected count (given \( x \)) of every arc
Running Example: Predict a Tagging

Lattice of exponentially many taggings (FST paths)
\[ p(path \mid x) = \frac{1}{Z(x)} \exp(\text{sum of } \theta \text{ values on } path) = \frac{1}{Z(x)} \text{ (product of } \phi \text{ values on } path) \]
where we define each \( \phi_k = \exp \theta_k \)

CRF replaces HMM’s probabilities by arbitrary “potentials” \( \phi_k = \exp \theta_k > 0 \), chosen to maximize \( \log p(y \mid x) \)
Summary of Probabilistic Methods in this Course!

- Each observed sentence $x$ has some unknown structure $y$.
- We like to define $p(x,y)$ as a product of quantities associated with different substructures of $(x,y)$.
  - These could be probabilities, or just potentials ($=$ exponentiated weights).
  - If they’re potentials, we’ll need to multiply $1/Z$ into the product too.
- Thus, $p(x) = \sum_y p(x,y)$ is a sum of products.
  - Among other uses, we need it to get $p(y | x) = p(x,y) / p(x)$.
  - Lots of summands, corresponding to different structures $y$, so we hope to do this sum by dynamic programming. (forward, inside algs)
- To increase $\log p(y_i | x_i)$ by stochastic gradient ascent or EM, must find substructures of $(x_i,y)$ that are expected under current $p(y | x_i)$.
  - More dynamic programming. (forward-backward, inside-outside algs)
- Simplest way to predict $y_i$ is as $\arg\max_y p(x_i,y)$.
  - More dynamic programming. (Viterbi inside, Viterbi forward)
  - This is actually all that’s needed to train a perceptron (not probabilistic).
  - Posterior decoding might get better results, depending on loss function.
From Log-Linear to Deep Learning: “Energy-Based” Models

Define \( p(y \mid x) = \frac{1}{Z(x)} \exp \text{score}(x, y) \)

Then

\[
\log p(y \mid x) = \text{score}(x, y) - \log Z(x)
\]

\[
\nabla \log p(y \mid x) = \nabla \text{score}(x, y) - \nabla \log Z(x)
\]

\[
= \nabla \text{score}(x, y) - \sum_{y'} p(y' \mid x) \nabla \text{score}(x, y')
\]

Log-linear case: \( \text{score}(x, y) = \vec{\theta} \cdot \vec{f}(x, y) \)

\[
\nabla_{\vec{\theta}} \log p(y \mid x) = \vec{f}(x, y) - \sum_{y'} p(y' \mid x) \vec{f}(x, y')
\]
Define $$p(y \mid x) = \frac{1}{Z(x)} \exp \text{score}(x, y)$$

- Why “energy-based”? These distributions show up in physics.
  - Let’s predict the 3-dimensional structure $$y$$ of a complex object $$x$$, such as a protein molecule
  - $$\text{energy}(x, y) =$$ a linear function: sums up local energies in the structure
  - Structure varies randomly according to **Boltzmann distribution**: $$p(y \mid x) = (1/Z(x)) \exp (-\text{energy}(x, y) / T)$$ where $$T > 0$$ is temperature
  - So at high temperature, many different structures are found; but as $$T \to 0$$, the protein is usually found in its lowest-energy shape.
  - In machine learning, we can define $$\text{energy}(x, y)$$ however we like.
Deep Scoring Functions

- Linear model: \( \text{score}(x, y) = \vec{\theta} \cdot \vec{f}(x, y) \)
  where \( f \) is a hand-designed feature vector

- What if we don’t want to design \( f \) by hand?
- Could we learn automatic features?

- Just define \( f \) using some additional parameters that have to be learned along with \( \theta \).
- \( \log p(y \mid x) \) may now have local maxima 😞
- We can still locally optimize it by stochastic gradient:
  \[
  \nabla \log p(y \mid x) = \nabla \text{score}(x, y) - \sum_{y'} p(y' \mid x) \nabla \text{score}(x, y')
  \]
Deep Scoring Functions

- **Linear model:**
  \[
  \text{score}(x, y) = \bar{\theta} \cdot \bar{f}(x, y)
  \]
  where \( f \) gets a hand-designed feature vector

- **Multi-layer perceptron:**
  \[
  f_k(x, y) = \begin{cases} 
  1 & \text{if } \bar{w}_k \cdot \bar{g}(x, y) > b_k \\
  0 & \text{otherwise}
  \end{cases}
  \]
  where \( g \) gives a (simple) hand-designed vector, maybe just basic attributes and embeddings. *(Or define \( g \) with same trick!)*
  \( f_k \) fires iff \( g(x,y) \) has enough of the attributes picked out by \( w_k \).
  So \( f_k \) is a kind of conjunctive feature. But by learning \( w_k \) and \( b_k \), we’re learning which attributes to conjoin and how strongly.
Deep Scoring Functions

- Linear model: \( \text{score}(x, y) = \bar{\theta} \cdot \bar{f}(x, y) \)

  where \( f \) gets a hand-designed feature vector

- Multi-layer perceptron: replace by something differentiable so we can learn by gradient ascent

\[
  f_k(x, y) = \begin{cases} 
  1 & \text{if } \bar{w}_k \cdot \bar{g}(x, y) > b_k \\
  0 & \text{otherwise}
  \end{cases}
\]
Deep Scoring Functions

- Linear model:  \( \text{score}(x, y) = \vec{\theta} \cdot \vec{f}(x, y) \)

  where \( f \) gets a hand-designed feature vector

- Neural network:  
  
  \[
  f_k(x, y) \approx \begin{cases} 
  1 & \text{if } \vec{w}_k \cdot \vec{g}(x, y) \gg b_k \\
  0 & \text{if } \vec{w}_k \cdot \vec{g}(x, y) \ll b_k 
  \end{cases}
  \]

  let  \( f_k(x, y) = \text{sigmoid}(\vec{w}_k \cdot \vec{g}(x, y) - b_k) \)

  where  \( \text{sigmoid}(s) = 1/(1 + \exp -s) \)
Deep Scoring Functions

- Linear model: \( \text{score}(x, y) = \bar{\theta} \cdot \bar{f}(x, y) \)

  where \( f \) gets a hand-designed feature vector

- Neural network:

  \[
  f_k(x, y) \approx \begin{cases} 
  1 & \text{if } \bar{w}_k \cdot \bar{g}(x, y) \gg b_k \\
  0 & \text{if } \bar{w}_k \cdot \bar{g}(x, y) \ll b_k 
  \end{cases}
  \]

- To define \( g(x,y) \), we might also learn “recurrent neural networks” that turn strings within \((x,y)\) into vectors.
  - E.g., the vector for “the tiny flea jumped” is the output of a neural network whose input concatenates the vector for “the tiny flea” (recursion!) with the embedding vector for “jumped”.
Why is discriminative training good?

- Perceptrons and CRFs and deep CRFs can efficiently make use of richer features.

Time flies like an arrow

Time flies like an arrow
Why is discriminative training good?

- And even with the same features, discriminative usually wins!
- Joint training tries to predict both $x$ and $y$.
- Discriminative training only tries to predict $y$ (given $x$), so it does a better job of that:
  - predict the correct $y$ (perceptron)
  - predict the distribution over $y$ (CRF)
- In fact, predicting $x$ and $y$ together may be too much to expect ...
Why is discriminative training good?

- Predicting $x$ and $y$ together may be too much to expect of a “weak model” like a PCFG or HMM.

- If you generate $(x,y)$ from a PCFG or HMM, it looks awful!
  - You get silly or ungrammatical sentences $x$.
  - Suggests that PCFG and HMM aren’t really such great models of $p(x,y)$, at least not at their current size ($\approx 50$ nonterminals or states).

- But generating $y$ given $x$ might still give good results.
  - PCFG and HMM can provide good conditional distributions $p(y \mid x)$.

- So just model $p(y \mid x)$. Twisting the weights to also predict sentences $x$ will distort our estimate of $p(y \mid x)$. 
Why is discriminative training good?

- Predicting $x$ and $y$ together may be too much to expect of a “weak model” like a PCFG or HMM.
- So just model $p(y | x)$. Twisting the weights to also predict sentences $x$ will distort our estimate of $p(y | x)$.
  
  - Let $p_\theta$ denote the PCFG with weight parameters $\theta$.
  - Joint training: Adjust $\theta$ so that $p_\theta(x,y)$ matches joint distribution of data.
  - Discrim. training: Adjust $\theta$ so that $p_\theta(y | x)$ matches conditional distribution.

Or equivalently, so that $p_{\text{hybrid}}(x,y)$ matches joint distribution:

$$p_{\text{hybrid}}(x,y) = p_{\text{empirical}}(x) \cdot p_{\theta}(y | x).$$

where $p_{\text{empirical}}(x) = 1/n$ for each of the $n$ training sentences, and is not sensitive to $\theta$. So we’re letting the data (not the PCFG!) tell us the distribution of sentences $x$. 

When do you want joint training?

- Predicting $x$ and $y$ together may be too much to expect of a “weak model” like a PCFG or HMM. So just model $p(y \mid x)$. Twisting the weights to also predict sentences $x$ will distort our estimate of $p(y \mid x)$.

- On the other hand, not trying to predict $x$ means we’re not learning from the distribution of $x$. “Throwing away data.”
  - Use joint training if we trust our model. Discriminative training throws away $x$ data only because we doubt we can model it well.
  - Also use joint in unsupervised/semisupervised learning. Here $x$ is all we have for some sentences, so we can’t afford to throw it away...
    - How can we know $y$ then? HMM/PCFG assumes $y$ latently influenced $x$.
    - EM algorithm fills in $y$ to locally maximize $\log p_\theta(x) = \log \sum_y p_\theta(x, y)$.
    - Requires joint model $p_\theta(x, y)$. (Q: Why not max $\log \sum_y p_\theta(y \mid x)$ instead?)
    - EM can work since the same $\theta$ is used to define both $p_\theta(x)$ and $p_\theta(y \mid x)$. Both come from $p_\theta(x, y)$ ($p_\theta(x) = \sum_y p_\theta(x, y)$ and $p_\theta(y \mid x) = p_\theta(x, y)/p_\theta(x)$). By observing $x$, we get information about $\theta$, which helps us predict $y$. 
Naïve Bayes vs. Logistic Regression

- Dramatic example of training $p(y \mid x)$ versus $p(x,y)$.

- Let’s go back to text categorization.
  - $x = (x_1, x_2, x_3, \ldots)$ (a feature vector)
  - $y = \{\text{spam, gen}\}$

- “Naïve Bayes” is a popular, very simple joint model:
  - $p(x,y) = p(y) \cdot p(x_1 \mid y) \cdot p(x_2 \mid y) \cdot p(x_3 \mid y) \cdot \cdots$
  - Q: How would you train this from supervised $(x,y)$ data?
  - Q: Given document $x$, how do we predict category $y$?
  - Q: What are the conditional independence assumptions?
  - Q: When are those “naïve” assumptions reasonable?
Naïve Bayes’s conditional independence assumptions break easily

- Pick $y$ maximizing $p(y) \cdot p(x_1 \mid y) \cdot p(x_2 \mid y) \cdot \ldots$
- $x =$ Buy this supercalifragilistic Ginsu knife set for only $39$ today ...
- Some features $x_k$ that fire on this example ...
  - Contains Buy
  - Contains supercalifragilistic
  - Contains a dollar amount under $100$
  - Contains an imperative sentence
  - Reading level = $7^{th}$ grade
  - Mentions money (use word classes and/or regexp to detect this)
  - ...
Naïve Bayes’s conditional independence assumptions break easily

- Pick $y$ maximizing $p(y) \cdot p(x_1 \mid y) \cdot p(x_2 \mid y) \cdot \ldots$
- $x = $ Buy this supercalifragilistic Ginsu knife set for only $\$39$ today …
- Some features $x_k$ that fire on this example, and their prob of firing when $y=\text{spam}$ versus $y=\text{gen}$:

<table>
<thead>
<tr>
<th></th>
<th>spam</th>
<th>gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>.5</td>
<td>.5</td>
<td>.02</td>
</tr>
<tr>
<td>.9</td>
<td>.1</td>
<td></td>
</tr>
</tbody>
</table>

- Contains a dollar amount under $\$100$
- 50% of spam has this – 25x more likely than in gen
- 90% of spam has this – 9x more likely than in gen
- Mentions money

Naïve Bayes claims $.5 \times .9 = 45\%$ of spam has both features – 25x9=225x more likely than in gen.

But here are the emails with both features – only 25x!
First feature implies second feature. Naïve Bayes is overconfident because it thinks they’re independent.
We have here a lousy model of \( p(x,y) \), namely \( p(y) \cdot p(x_1 | y) \cdot p(x_2 | y) \cdot \cdots \)

- If we used it to generate \((x,y)\), we'd get *incoherent* feature vectors that could not come from any actual document \( x \) ("mentions < $100"=1, "mentions money"=0).
- Its conditional distribution \( p(y | x) \) is nonetheless serviceable.

Training options:
- Supervised: maximize \( \log p(x,y) \). ("Naïve Bayes")
- Unsupervised: maximize \( \log p(x) = \log \sum_y p(x,y) \) via EM. ("document clustering")
- Supervised: maximize \( \log p(y | x) \). ("logistic regression")

Reinterpret Naïve Bayes conditional distrib as log-linear ("nuthin’ but adding weights"):

\[
\frac{p(y | x)}{p(x)} = \frac{p(x,y)}{p(x)} = \frac{(1/p(x)) \cdot p(y) \cdot p(x_1 | y) \cdot p(x_2 | y) \cdot \cdots}{Z(x)} = \frac{(1/Z(x)) \exp(\theta(y) + \theta(x_1, y) + \theta(x_2, y) \cdot \cdots)}{p(x)}
\]

where

\[
Z(x) = p(x) = \sum_y p(x,y) = \sum_y \frac{p(y) \cdot p(x_1 | y) \cdot p(x_2 | y) \cdot \cdots}{\theta(x,y)}
\]

\[
\theta(y) = \log p(y) \quad \theta(x_k, y) = \log p(x_k | y)
\]

So just do ordinary gradient ascent training of a conditional log-linear model.

Whose features are as shown: conjoin features of \( x \) with the identity of \( y \).
Logistic Regression doesn’t **model** \( x \), so doesn’t model \( x \)'s features as independent given \( y \)!

Naïve Bayes
\[
p(x_k \mid y) = \frac{p(y \mid x_k) p(x_k)}{p(y)}
\]

<table>
<thead>
<tr>
<th>Feature</th>
<th>( p(x_k \mid y) )</th>
<th>( \theta(x_k, y) ) after gradient ascent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains a dollar amount under $100</td>
<td>.5 .02</td>
<td>-1 -.85</td>
</tr>
<tr>
<td>Mentions money</td>
<td>.9 .1</td>
<td>-1 .15</td>
</tr>
</tbody>
</table>

Changed to compensate for the fact that whenever this feature fires, so will “Mentions money” feature.

Logistic regression trains weights to work together (needs gradient ascent). Naïve Bayes trains weights independently for each \( k \) (easier: count & divide).
Naïve Bayes
\[ p(x_k | y) \]

\[ \text{initial } \theta(x_k, y) = \log p(x_k | y) \]

\[ \text{final } \theta(x_k, y) \text{ after gradient ascent} \]

\begin{tabular}{c|c}
spam & gen \\
.5 & .02 \\
\hline
\end{tabular}

- Contains a dollar amount under $100

\begin{tabular}{c|c}
spam & gen \\
-1 & -5.6 \\
\hline
\end{tabular}

\begin{tabular}{c|c}
spam & gen \\
-.85 & -2.3 \\
\hline
\end{tabular}

.9 .1 - Mentions money

\begin{tabular}{c|c}
spam & gen \\
-.15 & -3.3 \\
\hline
\end{tabular}

**Q:** Is this truly just conditional training of the parameters of our original model? The old parameters were probabilities that had to sum to 1. But now it seems we’re granting ourselves the freedom to use any old weights that can no longer be interpreted as \( \log p(y) \) and \( \log p(x_k | y) \). Is this extra power why we do better?

**A:** No extra power! **Challenge:** Show how to adjust the weights after training, without disturbing \( p(y | x) \), to restore \( \sum_y \exp \theta(y) = 1 \) and \( \forall y \forall k \sum_{x_k} \exp \theta(x_k, y) = 1 \).
Summary

- Joint model, \( p(x,y) \):
  - Classical models: PCFG, HMM, Naïve Bayes
    - Product of many simple conditional distributions over generative moves.
    - “Locally normalized”: Each distribution must sum to 1: divide by some \( Z \).
  - Or Markov random field: \( p(x,y) = \frac{1}{Z} \exp \theta \cdot f(x,y) \)
    - “Globally normalized”: One huge distribution normalized by a single \( Z \).
    - \( Z \) is hard to compute since it sums over all parses of all sentences.

- Conditional model, \( p(y | x) \):
  - Conditional random field: \( p(y | x) = \frac{1}{Z(x)} \exp \theta \cdot f(x,y) \)
    - Globally normalized, but \( Z(x) \) only sums over all parses of sentence \( x \).
    - \( Z(x) \) is efficient to compute via inside algorithm.
    - Features can efficiently conjoin any properties of \( x \) and a “local” property of \( y \). Train by gradient ascent:
    - Doesn’t try to model \( p(x) \), i.e., “throws away” \( x \) data: good riddance?

- Discriminative model, \( \text{score}(x,y) \):
  - E.g., perceptron: No probabilistic interpretation of the score.
  - Train \( \theta \) to make the single correct \( y \) beat the others (for each \( x \)).
  - (Variants: Train to make “better” \( y \) values beat “worse” ones.)

Given \( x \), always compute best \( y \) by Viterbi algorithm. What’s different is the meaning of the resulting score.
Summary: When to build a generative \( p(x,y) \) vs. discriminative \( p(y|x) \) model?

- **Unsupervised learning?** \( \Rightarrow \) generative
  - Observing only \( x \) gives evidence of \( p(x) \) only.
  - Generative model says \( p(x) \) carries info about \( y \).
  - Discriminative model doesn’t care about \( p(x) \).
    - It only tries to model \( p(y|x) \), treating \( p(x) \) as “someone else’s job.”
    - So it will ignore our only training data as “irrelevant to my job.”

- (Intermediate option: contrastive estimation.)
Summary: When to build a generative $p(x,y)$ vs. discriminative $p(y|x)$ model?

- Unsupervised learning? $\Rightarrow$ generative
- Rich features? $\Rightarrow$ discriminative

😊 - Discriminative $p(y \mid x)$ can efficiently use features that consider arbitrary properties of $x$.
  - See earlier slides on “context-specific features.”
  - Also works for non-probabilistic discriminative models, e.g., trained by structured perceptron.

😊 - Generative $p(x,y)$ with the same features is usually too computationally hard to train.
  - Since $Z = \sum_{x,y} \theta f(x,y)$ would involve extracting features from every possible input $x$. 
Summary: When to build a generative $p(x,y)$ vs. discriminative $p(y|x)$ model?

- Unsupervised learning? → generative
- Rich features? → discriminative

Neither case? → let dev data tell you!

- Use a generative model, but choose $\theta$ to max
  $\log p_{\theta}(y|x) + \lambda \log p_{\theta}(x) + c$ Regularizer($\theta$)

- Tune $\lambda$, $c$ on dev data
  - $\lambda = 0$ → discriminative training (best to ignore distrib of $x$)
  - $\lambda = 1$ → generative training (distrib of $x$ gives useful info about $\theta$: true if the data were truly generated from your model!)
  - $\lambda = 0.3$ → in between (distrib of $x$ gives you some useful info but trying too hard to match it would harm predictive accuracy)
Summary: When to build a generative $p(x,y)$ vs. discriminative $p(y|x)$ model?

- Unsupervised learning? ➔ generative
- Rich features? ➔ discriminative

Neither case? ➔ let dev data tell you!

- Use a generative model, but train $\theta$ to max
  $$\log p_\theta(y|x) + \lambda \log p_\theta(x) + c \text{ Regularizer}(\theta)$$
- What you really want is high $\log p_\theta(y|x)$ on future data.
- But you only have a finite training sample to estimate that. Both blue terms provide useful bias that can help compensate for the variance of your estimate.
- Same idea as multi-task or multi-domain learning: to find params that are good at your real task (predicting $y$ from $x$), slightly prefer params that are also good at something related (predicting $x$).
Summary: When to build a generative $p(x,y)$ vs. discriminative $p(y|x)$ model?

- Unsupervised learning? ➔ generative
- Rich features? ➔ discriminative

Neither case? ➔ let dev data tell you!

- Use a generative model, but train $\theta$ to max
  \[ \log p_\theta(y|x) + \lambda \log p_\theta(x) + c \text{ Regularizer}(\theta) \]

- Note: Equivalent to
  \[
  (1-\lambda) \log p_\theta(y|x) + \lambda \log p_\theta(y|x) + \lambda \log p_\theta(x) + c \text{ R}(\theta)
  = (1-\lambda) \log p_\theta(y|x) + \lambda \log p_\theta(x,y) + c \text{ R}(\theta)
  \]

- So on each example, stochastic gradient ascent can stochastically follow the gradient of discriminative $\log p_\theta(y|x)$ with prob $1-\lambda$ or generative $\log p_\theta(x,y)$ with prob $\lambda$
Summary: When to build a generative $p(x,y)$ vs. discriminative $p(y|x)$ model?

- Use a generative model, but train $\theta$ to max
  $\log p_\theta(y|x) + \lambda \log p_\theta(x) + c$ Regularizer($\theta$)
- What you really want is high $\log p_\theta(y|x)$ on future data.

Ok, maybe not quite. Suppose you will act at test time (and dev time) using a decision rule $\delta_\theta(x)$ that tells you what action to take on input $x$, using the learned parameters $\theta$. And the loss function $L(a \mid x,y)$ tells you how bad action $a$ would be in a situation with a particular input $x$ if the unobserved output were $y$. Then what you really want is low loss on future data: $L(\delta_\theta(x) \mid x,y)$ should be low on average. So replace $\log p_\theta(y|x)$ in training with $-L(\delta_\theta(x) \mid x,y)$, or perhaps $-L(\delta_\theta(x) \mid x,y) + \lambda' \log p_\theta(y|x)$.

Decision rules don’t have to be probabilistic or differentiable: e.g., the perceptron uses a linear scoring function as its decision rule. (At training time it simply tries to drive training loss to 0.) But if you have a good probability model $p_\theta(y \mid x)$, then the ideal decision rule is minimum Bayes risk (MBR): $\delta_\theta(x) = \text{argmin}_a \sum_y p_\theta(y \mid x) \ L(a \mid x,y)$. (Risk means expected loss.) MBR reduces to the Viterbi decision rule, $\delta_\theta(x) = \text{argmax}_y p_\theta(y \mid x)$, in the special case where actions are predictions of $y$ and we use “0-1” loss, that is, $L(a \mid x,y)=(\text{if } (a==y) \text{ then 0 else 1})$. Posterior decoding is the MBR rule for a different loss function: it chooses a tag sequence $a$ that minimizes the expected number of incorrect tags (possibly $p(a \mid x)=0$, unlike Viterbi!).