

Probabilistic Models for Computational Linguistics

Mark Johnson

Macquarie University

HCSNet Winterfest 2010

Outline

Overview

n-gram language modelling

Machine translation

Sequence tagging with Hidden Markov Models

Grammars and Syntactic Parsing

PCFGs and beyond for statistical parsing

The big ideas

- Probabilistic generative models
- The *noisy channel* model, and recovering hidden structure
- The *Expectation Maximisation* (EM) algorithm
- *Hidden Markov Models* (HMMs)
- *Probabilistic Context-Free Grammars* (PCFGs)
- *Maximum Entropy* (MaxEnt) models

Brief Introduction to Discrete Probability

- *Sample space* Ω : set of possible events
- *Probability* $P(x)$ of event $x \in \Omega$
 - ▶ $0 \leq P(x) \leq 1$ for all $x \in \Omega$
 - ▶ $\sum_{x \in \Omega} P(x) = 1$
- A *random variable* is a function on Ω
 - ▶ If A is a random variable and a is one of its values,

$$P(A = a) = \sum_{\substack{x \in \Omega \\ A(x) = a}} P(x)$$

i.e., $P(A = a)$ is sum of probability of all events x where $A(x) = a$.

- **Example:** $\Omega = \{\text{Sam, Sandy, likes}\}$
 $P(\text{Sam}) = 0.5, P(\text{Sandy}) = 0.1, P(\text{likes}) = 0.4$
 $PoS(\text{Sam}) = n, PoS(\text{Sandy}) = n, PoS(\text{likes}) = v$
 $P(PoS = n) = 0.6, P(Pos = v) = 0.4$

Joint and conditional probabilities

- *Joint probability* of two random variables A and B is:

$$P(A = a, B = b) = \sum_{\substack{x \in \Omega \\ A(x) = a \\ B(x) = b}} P(x)$$

is sum of probability of all events x where $A(x) = a$ and $B(x) = b$

- *Conditional probability* of A given B is:

$$P(A = a \mid B = b) = \frac{P(A = a, B = b)}{P(B = b)}$$

- Example: $\Omega = \{\text{Sam, Sandy, likes}\}$

$$P(\text{Sam}) = 0.5, P(\text{Sandy}) = 0.1, P(\text{likes}) = 0.4$$

$\text{Len}(x)$ = number of chars in x

$$P(\text{PoS} = n \mid \text{Len} = 5) = \frac{P(\text{PoS} = n, \text{Len} = 5)}{P(\text{Len} = 5)} = 0.2$$

Bayes rule

- For any random variables A, B :

$$P(A, B) = P(A | B) P(B) = P(B | A) P(A)$$

so:

$$P(A | B) = \frac{P(B | A) P(A)}{P(B)}$$

- Bayes rule is useful for *inverting conditional probability distributions*

The Noisy Channel model and Bayes rule

- The *Noisy Channel model* uses Bayes rule to invert probability distributions
- Example: Speech recognition maps *acoustic inputs* A to *texts* T
- Probabilistic formulation:

$$\begin{aligned} T^*(A) &= \operatorname{argmax}_T P(T | A) \\ &= \operatorname{argmax}_T \frac{P(A | T) P(T)}{P(A)} \\ &= \operatorname{argmax}_T P(A | T) P(T) \end{aligned}$$

- $P(T)$ is called the *source model*
- $P(A | T)$ is called the *channel model*
- The noisy channel model is called a *generative model* because it defines a joint probability distribution over its variables
- Other applications of noisy channel models:
 - ▶ machine translation
 - ▶ language reconstruction in historical linguistics
 - ▶ detecting and correcting disfluencies in speech recognition



Outline

Overview

n-gram language modelling

Machine translation

Sequence tagging with Hidden Markov Models

Grammars and Syntactic Parsing

PCFGs and beyond for statistical parsing

Language modelling overview

- Goal of language modelling: distinguish more likely text or speech from less likely text or speech
- A key component of many NLP systems
 - ▶ Speech recognition: e.g., “recognise speech” vs “wreck a nice beach”
 - ▶ Machine translation
- n -gram models model text in terms of overlapping n -word sequences
 - ▶ easy to train from billions of words of text
 - ▶ produce quite good results
- They also introduce key notions of probability and statistics

What is a language model?

- A sentence or a document is a *sequence of words* $\mathbf{w} = (w_1, \dots, w_n)$, where each $w_i \in \mathcal{W}$ (i.e., \mathcal{W} is the vocabulary)
- Goal: find an accurate probability distribution $P(\mathbf{w})$ over word sequences
- Method: *estimate* $P(\mathbf{w})$ using *statistics* collected from a *corpus* of sentences
 - ▶ a *statistic* is a function of the data
- Questions:
 - ▶ which statistics should we collect?
 - ▶ how do we use them to estimate $P(\mathbf{w})$?
- Why is this hard?

“Bag of words” (unigram) models

- “Bag of words” assumption: for all positions i, j in a sentence

$$P(W_i = w) = P(W_j = w)$$

- To generate a sentence $\mathbf{w} = (w_1, \dots, w_n)$:
 1. generate a *sentence length* n from $P(N)$
 2. generate word i from a word distribution $P(W_i)$

$$P(\mathbf{W} = (w_1, \dots, w_n)) = P(N = n) \prod_{i=1}^n P(W_i = w_i)$$

- A unigram model lets us define a distribution over sentences in terms of distributions over sentence lengths and distributions over words

Estimating $P(W)$ from a corpus

- Informal idea:
 - ▶ Given a corpus D , count how often each word w appears
 - ▶ Set $P(W = w)$ to the fraction of times w appears in D
- Formalise as a *parameter estimation problem*
 - ▶ Introduce parameters

$$P(W_i = w) = \varphi_w \text{ for all } w \in \mathcal{W} \text{ and } i = 1, 2, \dots$$

so φ is a *vector* indexed by *word types* in \mathcal{W}

- ▶ Provide an *estimator* $\hat{\varphi}(D)$ for the parameters from data D

$$\hat{\varphi}_w = \frac{n_w(D)}{n(D)}, \text{ where:}$$

$n_w(D)$ = number of times w appears in D , and

$$n.(D) = \sum_{w \in \mathcal{W}} n_w(D) = \text{number of words in } D$$

Maximum likelihood estimation

- *Maximum likelihood principle:*

Choose model parameters to make data *as likely as possible*

- *Likelihood function:* probability of data *as function of model parameters*
- If $D = (w_1, \dots, w_n)$ (i.e., one long sentence)

$$L_D(\varphi) = P_\varphi(D) = P(N = n) \prod_{i=1}^n \varphi_{w_i}$$

- Maximum Likelihood Estimate (MLE):

$$\hat{\varphi} = \underset{\varphi}{\operatorname{argmax}} L_D(\varphi)$$

- Fact: MLE is relative frequency, i.e.,

$$\hat{\varphi}_w = \frac{n_w(D)}{n.(D)}$$

Smoothing the MLE $\hat{\varphi}$

- Suppose w contains an *unknown word* w , i.e., $n_w(D) = 0$
 - $\Rightarrow \hat{\varphi}_w = 0$
 - $\Rightarrow P_{\varphi}(w) = 0$

\Rightarrow Use a different estimator, e.g., *Bayesian MAP estimator*

$$\tilde{\varphi}_w = \frac{n_w(D) + 1}{n.(D) + m}, \text{ where:}$$

$$m = |\mathcal{W}| = \text{size of vocabulary}$$

- The Bayesian MAP estimator *smooths* the MLE

More on unknown words

- In many applications you'll often encounter unknown words, no matter how big training data D is
 - One approach: replace all words not seen (say) 5 times in D with a special symbol, say $\star\text{UNK}\star$
 - Problem: for many infrequent words w , usually $\varphi_{\star\text{UNK}\star} > \varphi_w$
 - More sophisticated models of unknown words make a big difference
 - “Lazy statistical modeller’s morphology”
 - ▶ Multiple kinds of “unknown words”
 - ▶ classify unknown words by
 - capitalisation, or
 - last three letters
- e.g., Melbourne $\mapsto \star\text{UNK}_{\text{CAP}}\star$, walking $\mapsto \star\text{UNK}_{\text{ing}}\star$

Bigram language model

- A *bigram language model* captures dependencies between adjacent words
- Conditional decomposition: (exact)

$$\begin{aligned}P(w_1, \dots, w_n) &= P(w_1, \dots, w_{n-1}) P(w_n \mid w_1, \dots, w_{n-1}) \\ &= P(w_1) P(w_2 \mid w_1) P(w_3 \mid w_1, w_2) \dots \\ &\quad P(w_n \mid w_1, \dots, w_{n-1})\end{aligned}$$

- *Markov*: only last word matters (approximation)

$$P(w_i \mid w_1, \dots, w_{i-1}) = P(w_i \mid w_{i-1}), \text{ so:}$$

$$P(w_1, \dots, w_n) = P(w_1) P(w_2 \mid w_1) P(w_3 \mid w_2) \dots P(w_n \mid w_{n-1})$$

- *Stationarity*: position doesn't matter (approximation)

$$P(w_i \mid w_{i-1}) = P(w_j \mid w_{j-1}) \text{ for all } i, j$$

Sentence boundary markers

- Instead of generating w_1, \dots, w_n , generate the infinite sequence

$$\triangleright, w_1, \dots, w_n, \triangleleft, \triangleleft, \dots$$

where $w_0 = \triangleright$ is the *beginning of sentence marker*

and $w_{n+1} = \triangleleft$ is the *end of sentence marker*

⇒ Everything is a bigram dependency, e.g., $P(w_1) = P(w_1 | \triangleright)$;
 $P(\triangleleft | \triangleleft) = 1$

- Bigram language model:

$$\begin{aligned} P(w_1, \dots, w_n) &= \prod_{i=1}^{n+1} P(w_i | w_{i-1}) \\ &= \prod_{i=1}^{n+1} \theta_{w_i, w_{i-1}} \end{aligned}$$

where $\theta_{w, w'} = P(W_i = w | W_{i-1} = w')$ (stationarity)

- θ is a matrix of transition probabilities indexed by \mathcal{W} .

Bigram example

- Bigram parameters:

$\theta =$	$w_i \backslash w_{i-1}$	\triangleright	likes	Sam	Sandy
	likes	0.1	0.1	0.2	0.4
	Sam	0.5	0.2	0.1	0.1
	Sandy	0.4	0.6	0.1	0.1
	\triangleleft	0	0.1	0.6	0.4

- Bigram model estimates:

$$\begin{aligned} P(\triangleright, \text{Sam}, \triangleleft) &= \theta_{\text{Sam}, \triangleright} \theta_{\triangleleft, \text{Sam}} \\ &= 0.5 \times 0.6 \end{aligned}$$

$$\begin{aligned} P(\triangleright, \text{Sandy}, \text{likes}, \text{Sam}, \triangleleft) &= \theta_{\text{Sandy}, \triangleright} \theta_{\text{likes}, \text{Sandy}} \theta_{\text{Sam}, \text{likes}} \theta_{\triangleleft, \text{Sam}} \\ &= 0.4 \times 0.4 \times 0.2 \times 0.6 \end{aligned}$$

Estimating bigram model parameters θ

- Maximum Likelihood Estimate from corpus D :

$$\hat{\theta}_{w,w'} = \frac{n_{w,w'}(D)}{n_{\cdot,w'}(D)}, \text{ where:}$$

$n_{w,w'}(D)$ = number of times w follows w' in D , and

$n_{\cdot,w'}(D)$ = number of times w' follows anything in D

- Data sparsity is even more problematic for bigram models
- Many methods for *smoothing using a unigram model* $\hat{\varphi}$, e.g.:

$$\tilde{\theta}_{w,w'} = \lambda \hat{\varphi}_w + (1 - \lambda) \hat{\theta}_{w,w'}$$

- Choose *interpolation parameter* λ to maximize likelihood of a *heldout corpus*

Further work in language modelling

- Standard to work with 3-gram to 5-gram models estimated from billions of words of text
- Smoothing is essential, and the method used makes a big difference
- Wide variety of language models have been investigated
 - ▶ *Trigger language models* try to track the change in vocabulary within a document
 - ▶ *Syntactic (i.e., parsing-based) language models* typically outperform n -grams when *trained on same sized corpora*

Outline

Overview

n -gram language modelling

Machine translation

Sequence tagging with Hidden Markov Models

Grammars and Syntactic Parsing

PCFGs and beyond for statistical parsing

Machine translation overview

- A *machine translation system* automatically translates text or speech from one language to another
- Uses the noisy channel to decompose the problem into language model and translation model
- Introduces the *Expectation Maximisation* (EM) algorithm for learning from *hidden data*
 - ▶ translation model maps *words to words* or *phrases to phrases*
 - ▶ most naturally learnt from *word-aligned* parallel translations
 - ▶ but virtually all training data is *sentence-aligned*⇒ word alignments are *hidden*
- EM learns *word alignments* and *word translation probabilities* from sentence-aligned data

Noisy channel model for translation

- Goal: translate a sentence f (e.g., “foreign”, “French”) into a sentence e (“English”)

$$E^*(f) = \operatorname{argmax}_e P(e | f)$$

- Noisy channel model (i.e., Bayes inversion)

$$\operatorname{argmax}_e P(e | f) = \operatorname{argmax}_e P(e) P(f | e)$$

- $P(e)$ is a *language model*, typically trained on *billions* of words of monolingual text
- $P(f | e)$ is a *channel model*, typically trained on *millions* of words of parallel text

Channel models for machine translation

- Overview of machine translation channel model:
 - ▶ Analyse the English input e into a sequence of words or phrases
 - ▶ Replace each English word or phrase with its French translation
 - ▶ Reorder the translated words or phrases to produce French output f
- Translation units and reorderings can be identified in many ways
 - ▶ here the translation units are *words*
 - ▶ initially we will assume *every reordering is equally likely*
 - ▶ but it's easy to use our methods to learn *string position based reordering*

The IBM 1 translation model $P(\mathbf{f} | \mathbf{e})$

- A *word alignment* \mathbf{a} pairs each French word f_k to a single English word e_{a_k} .
- One English word may be aligned with several French words
- We add a special “null word” \diamond to e to pair French words with no English counterpart
- To generate \mathbf{f} from \mathbf{e} :
 - ▶ Pick an alignment \mathbf{a}
 - ▶ For each k , generate f_k from e_{a_k} according to $P(f_k | e_{a_k})$

Position	e	\mathbf{a}	\mathbf{f}
0	\diamond		
1	I	→	Je
2	did	↗	ne
3	n't	↘	mange
4	eat	↘	pas
5	bread	↘	du
6		↘	pain

$\mathbf{a} = (1, 3, 4, 3, 0, 6)$

Mathematical formulation of IBM model 1

- Simple generative model:

$$\begin{aligned}P(\mathbf{a}, \mathbf{f} | \mathbf{e}) &= P(\mathbf{a} | \mathbf{e}) P(\mathbf{f} | \mathbf{a}, \mathbf{e}) \\ &= P(\mathbf{a} | \mathbf{e}) \prod_{k=1}^{|\mathbf{a}|} P(f_k | e_{a_k}), \text{ so:}\end{aligned}$$

$$P(\mathbf{f} | \mathbf{e}) = \sum_{\mathbf{a}} P(\mathbf{a} | \mathbf{e}) \prod_{k=1}^{|\mathbf{a}|} P(f_k | e_{a_k})$$

- In our simple generative model, $P(\mathbf{a} | \mathbf{e})$ is a *uniform distribution* over alignments given the French sentence length $|\mathbf{f}|$
- Even so, $P(\mathbf{a} | \mathbf{e}, \mathbf{f})$ is *not uniform!*

$$\begin{aligned}P(\mathbf{a} | \mathbf{e}, \mathbf{f}) &= \frac{P(\mathbf{a}, \mathbf{f} | \mathbf{e})}{P(\mathbf{f} | \mathbf{e})} \\ &\propto \prod_{k=1}^{|\mathbf{a}|} P(f_k | e_{a_k})\end{aligned}$$

Learning translation probabilities from a word-aligned corpus

- Goal: learn word translation probabilities $P(f | e)$
- Easy with a *word-aligned parallel corpus* $D = (e, f, a)$
- Let $P(f | e) = \tau_{f,e}$. Then e.g., the MLE is:

$$\hat{\tau}_{f,e} = \frac{n_{f,e}(a)}{n_{\cdot,e}(a)}, \text{ where:}$$

$n_{f,e}(a)$ = the number of times f is aligned to e in D , and

$$n_{\cdot,e}(a) = \sum_f n_{f,e}(a)$$

= the number of times e is aligned to anything in D

- But word-aligned corpora are very expensive
can we learn translation probabilities from sentence-aligned corpora instead?

The Expectation Maximization Algorithm

- EM is useful for problems
 - ▶ with *hidden variables* (variables whose values are unknown, e.g., alignments \mathbf{a}), and
 - ▶ where estimation would be easy if those variables' values were known
- Informal description of EM algorithm:

Initialise model parameters $\tau^{(0)}$ somehow (e.g., randomly)

Repeat for $t = 0, 1, \dots$ until convergence:

- *E-step*: Compute *expected value* of sufficient statistics

$$E_{\tau^{(t)}}[n_{f,e}] = \sum_{\mathbf{a}} n_{f,e}(\mathbf{a}) P_{\tau^{(t)}}(\mathbf{a} \mid e, \mathbf{f})$$

- *M-step*: Compute $\tau^{(t+1)}$ from $E[n_{f,e}]$

$$\tau_{f,e}^{(t+1)} = \frac{E_{\tau^{(t)}}[n_{f,e}(\mathbf{a})]}{E_{\tau^{(t)}}[n_{\cdot,e}(\mathbf{a})]}$$

- Every iteration of EM is *guaranteed* to increase the likelihood of data, but it is not guaranteed to converge to the MLE

Expected values of translation counts

- If g is a real-valued random variable, its *expected value* $E[g]$ is:

$$E[g] = \sum_{x \in \Omega} g(x) P(x)$$

i.e., an expected value is a kind of weighted average

- If we're given an alignment \mathbf{a} , we can calculate translation counts $n_{f,e}(\mathbf{a})$ directly
- If we're given a probability distribution $P(\mathbf{a})$ over alignments, we can calculate the *expected translation counts*:

$$E[n_{f,e}] = \sum_{\mathbf{a}} n_{f,e}(\mathbf{a}) P(\mathbf{a} | \mathbf{e}, \mathbf{f})$$

A single iteration of the EM algorithm

- At the beginning of iteration t , the current estimate of the word translation probabilities is $\tau^{(t)}$:

1. *E-step*:

- a. For each possible alignment \mathbf{a} , calculate $P_{\tau^{(t)}}(\mathbf{a} \mid e, \mathbf{f})$
- b. Summing over all possible alignments \mathbf{a} , calculate expected translation counts

$$E_{\tau^{(t)}}[n_{f,e}] = \sum_{\mathbf{a}} n_{f,e}(\mathbf{a}) P_{\tau^{(t)}}(\mathbf{a} \mid e, \mathbf{f})$$

2. *M-step*: Use the expected counts to update τ :

$$\tau_{f,e}^{(t+1)} = \frac{E_{\tau^{(t)}}[n_{f,e}]}{\sum_f E_{\tau^{(t)}}[n_{f,e}]}$$

- Unfortunately, the algorithm as described is infeasible ...

Dynamic programming for expected counts

- By definition, every French word f_k in the corpus is aligned with some English word e_{a_k} , but *which one*?
- Because we assume $P_{\tau}(a | e)$ is uniform, alignment probability only depends on τ . With some algebra, can show:

$$P_{\tau}(a_k = j | e, f_k) = \frac{\tau_{f_k, e_j}}{\sum_{j'=1}^{|e|} \tau_{f_k, e_{j'}}$$

- Expected translation counts:

$$E_{\tau}[n_{f', e'}] = \sum_{\substack{k: f_k = f' \\ j: e_j = e'}} P_{\tau}(a_k = j | e, f_k)$$

- Informally, to compute transition counts iterate through each French word f_k in parallel corpus
 - ▶ compute probability $P_{\tau}(a_k = j | e, f_k)$ that it is aligned to e_j
 - ▶ add $P_{\tau}(a_k = j | e, f_k)$ to expected counts $E_{\tau}[n_{f_k, e_j}]$

EM algorithm for learning alignments and translation probabilities

Initialise translation probabilities τ somehow (e.g., randomly)

Repeat for $t = 0, 1, \dots$ until convergence:

Initialise expected counts $E[n_{f,e}] = 0$ for all e, f

For each French word position $k \in 1, \dots, |\mathbf{f}|$:

For each English word position $j \in 1, \dots, |\mathbf{e}|$:

$$E[n_{f_k, e_j}] \quad + = \quad P_{\tau}(a_k = j \mid e, f_k)$$

Recalculate translation probabilities τ :

$$\tau_{f,e} = \frac{E[n_{f,e}]}{E[n_{.,e}]}$$

From IBM model 1 to a machine translation system

- IBM model 1 produces surprisingly accurate alignments and word translation probabilities
- For translation you need a nontrivial *alignment model* $P(\mathbf{a} | e)$
- Increasing interest in using syntax to define phrasal translation units and reordering models
- The problem of finding the optimal translation is usually NP-hard
 - ▶ beam search for optimal translation

Outline

Overview

n -gram language modelling

Machine translation

Sequence tagging with Hidden Markov Models

Grammars and Syntactic Parsing

PCFGs and beyond for statistical parsing

Sequence tagging overview

- Sequence tagging:
given a sequence (x_1, \dots, x_n) of *observations* (where each $x_i \in \mathcal{X}$),
return a sequence (y_1, \dots, y_n) of *states* or *labels* (where each $y_i \in \mathcal{Y}$)
- Hidden Markov Models, and their three different representations:
 - ▶ stochastic automata
 - ▶ Bayes nets
 - ▶ Trellis
- Even though there are *exponentially many label sequences*, there are *linear time dynamic programming algorithms*
 - ▶ *Viterbi algorithm* for finding most likely label sequence
 - ▶ *Forward-backward algorithm* for finding expected counts
- Expectation Maximisation algorithm for learning HMMs from strings alone

Applications of sequence tagging

- *Part of speech (POS) tagging*: x are the words of a sentence, y are their parts of speech.

y : DT JJ NN VBD NNP .
 x : the big cat bit Sam .

- *Noun-phrase chunking*: x are the words of a sentence, y indicates whether they are in the beginning, middle or end of a noun phrase (NP) chunk.

y : [NP NP NP] - [NP] .
 x : the big cat bit Sam .

- *Named entity detection*: The x are the words of a sentence, y indicates whether they are in the beginning, middle or end of a noun phrase (NP) chunk that is the name of a person, company or location.

y : [CO CO] - [LOC] - [PER] -
 x : XYZ Corp. of Boston announced Spade's resignation

- *Speech recognition*: x are 100 msec. time slices of acoustic input, and y are the corresponding phonemes (i.e., y_i is the phoneme being uttered in time slice x_i)

Hidden Markov Models (HMMs)

- A *Markov model* generates sequence $\mathbf{y} = (y_0 = \triangleright, y_1, \dots, y_n, y_{n+1} = \triangleleft)$ by generating each y_i from y_{i-1}

$$P(\mathbf{y}) = \prod_{i=1}^{n+1} P(y_i | y_{i-1}) = \prod_{i=1}^{n+1} \sigma_{y_i, y_{i-1}}$$

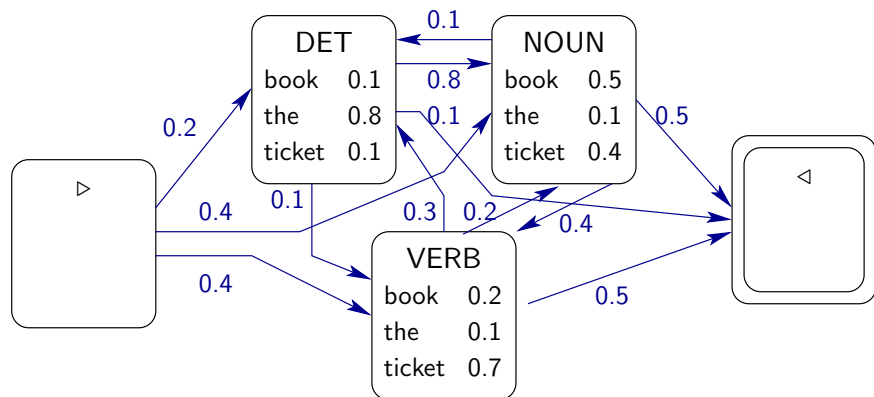
- In a *Hidden Markov Model*:
 - ▶ the *state sequence* $\mathbf{y} = (y_1, \dots, y_n)$ is generated by a Markov model
 - ▶ each x_i in the *observation sequence* $\mathbf{x} = (x_1, \dots, x_n)$ is generated from its corresponding state y_i

$$P(\mathbf{x} | \mathbf{y}) = \prod_{i=1}^n P(x_i | y_i) = \prod_{i=1}^n \tau_{x_i, y_i}$$

- ▶ so the joint distribution over (\mathbf{x}, \mathbf{y}) is:

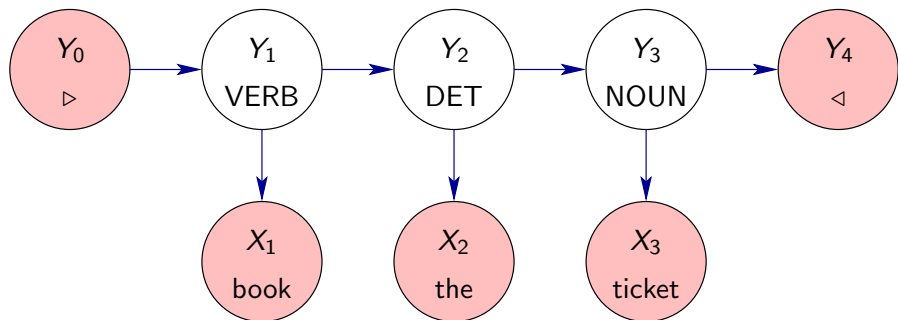
$$P(\mathbf{x}, \mathbf{y}) = \left(\prod_{i=1}^n P(y_i | y_{i-1}) P(x_i | y_i) \right) P(\triangleleft | y_n) = \left(\prod_{i=1}^n \sigma_{y_i, y_{i-1}} \tau_{x_i, y_i} \right) \sigma_{\triangleleft, y_n}$$

HMMs as stochastic automata (Moore machines)



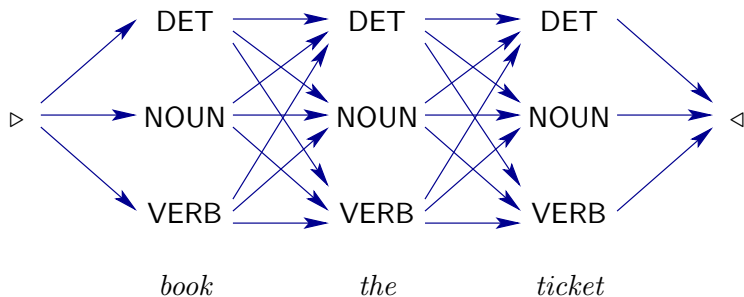
- Automaton depicts all possible (x, y) sequences
- States are nodes in graph (identified with a label)
- State-to-state transition probabilities σ are arc labels
- State-to-observation emission probabilities τ are node labels

HMMs as Bayes nets



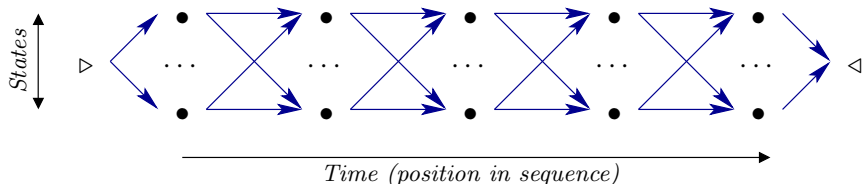
- Each random variable (e.g., observation x_i and label y_i) is represented by a node
- Shows factorisation of joint distribution $P(\mathbf{x}, \mathbf{y})$ into product of conditional distributions
- Each node is associated with a conditional distribution from each of its parents

HMM computation as a Trellis



- The *trellis* represents *all possible state sequences* $\mathbf{y} \in \mathcal{Y}^n$ for a fixed *observation sequence* $\mathbf{x} = (x_1, \dots, x_n)$
- One node for each (y, i) combination for $y \in \mathcal{Y}$, $i \in 1, \dots, n$
- The *weight of an edge* from $(y', i - 1)$ to (y, i) is $\sigma_{y,y'}\tau_{x_i,y}$, i.e., *probability of moving from y' to y and emitting x_i*
- The *weight of a path* is the *product* of the weights of the edges in it
- Each state sequence \mathbf{y} identifies a path from \triangleright to \triangleleft in trellis
- $P(\mathbf{x}, \mathbf{y}) = \text{weight of path } \mathbf{y}$

Calculating the probability of an observation sequence x

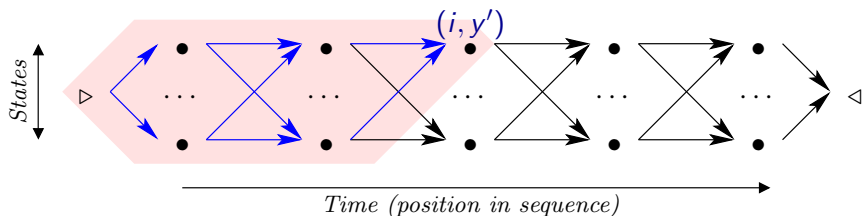


- Goal: compute probability $P_{\sigma, \tau}(x)$ of observed sequence x

$$\begin{aligned} P_{\sigma, \tau}(x) &= \sum_{\mathbf{y} \in \mathcal{Y}^n} P_{\sigma, \tau}(x, \mathbf{y}) \\ &= \sum_{\mathbf{y} \in \mathcal{Y}^n} \left(\prod_{i=1}^n \sigma_{y_i, y_{i-1}} \tau_{x_i, y_i} \right) \sigma_{\triangleleft, y_n} \end{aligned}$$

- $P_{\sigma, \tau}(x)$ is the *sum of the probability of all paths* in trellis
 - The number of sequences in \mathcal{Y}^n grows exponentially with n
- \Rightarrow *calculating $P_{\sigma, \tau}(x)$ by direct summation is infeasible*

Forward probabilities α

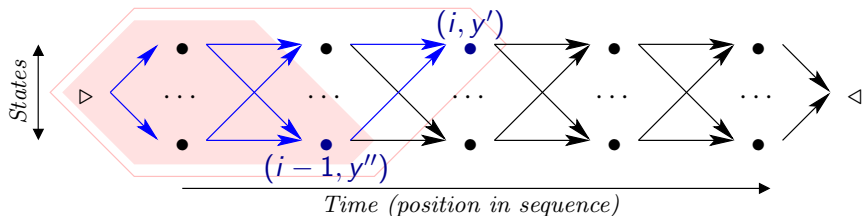


- **Forward probability** $\alpha_{i,y'}$ is the probability of generating (x_1, \dots, x_i) ending in a state y'

$$\alpha_{i,y'} = \sum_{\mathbf{y} \in \mathcal{Y}^{i-1}} P_{\sigma, \tau}(Y_{1:i-1} = \mathbf{y}, Y_i = y', V_{x_{1:i}})$$

- $\alpha_{i,y'}$ is sum of weights of all paths from \triangleleft to (i, y')
- α is an array indexed by a *position* i and a state $y' \in \mathcal{Y}$
- Probability of observations $P_{\sigma, \tau}(\mathbf{x}) = \alpha_{n+1, \triangleleft}$

Dynamic programming for forward probabilities α



- With simple algebra it's possible to show that:

$$\alpha_{i,y'} = \sum_{y'' \in \mathcal{Y}} \alpha_{i-1,y''} (\sigma_{y',y''} \tau_{x_i,y'})$$

\Rightarrow *Dynamic programming algorithm for calculating α*

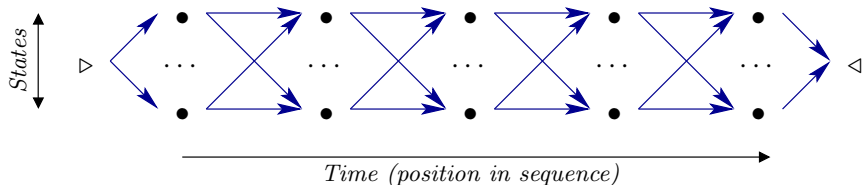
Set $\alpha_{0,\triangleleft} = 1$ and $\alpha_{0,y} = 0$ for all other $y \in \mathcal{Y}$

For $i = 1, \dots, n$:

For $y' \in \mathcal{Y}$:

$$\alpha_{i,y'} = \sum_{y'' \in \mathcal{Y}} \alpha_{i-1,y''} (\sigma_{y',y''} \tau_{x_i,y'})$$

Finding the most likely state sequence

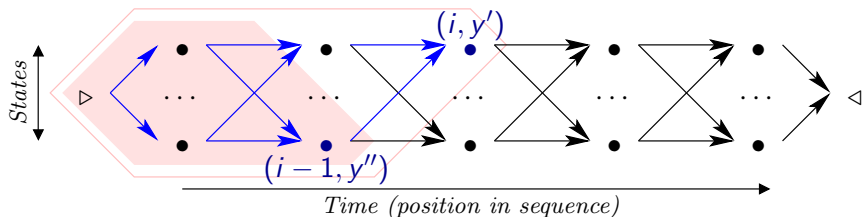


- Goal: find the most likely state sequence $\mathbf{Y}^*(\mathbf{x})$ wrt HMM (σ, τ)

$$\mathbf{Y}^*(\mathbf{x}) = \underset{\mathbf{y} \in \mathcal{Y}^n}{\operatorname{argmax}} P(\mathbf{y} | \mathbf{x}) = \underset{\mathbf{y}}{\operatorname{argmax}} P(\mathbf{y}, \mathbf{x})$$

- ▶ This is how you use an HMM to tag an observation sequence \mathbf{x}
- Key observation: *best path from \triangleright to (i, y') must begin with the best path to $(i - 1, y'')$ for some $y'' \in \mathcal{Y}$*

The Viterbi algorithm for most likely state sequence



- Compute best path to every node (i, y') as well as its weight
- Algorithm overview:

For $i = 1, \dots, n + 1$:

for each $y' \in \mathcal{Y}$:

calculate best path to (i, y') and its probability

using best paths to $(i-1, y'')$ and their probabilities

Estimating HMMs from *visible* training data

- Training data $D = (\mathbf{x}, \mathbf{y}) = ((x_1, \dots, x_n), (y_1, \dots, y_n))$, i.e., observations and their states
 - ▶ E.g., *They/PRP neither/CC liked/VBD nor/CC disliked/VBD the/DT Old/NNP Man/NNP*
- Because HMMs are products of multinomials, the MLEs are *relative frequencies*

$$\hat{\sigma}_{y,y'} = \frac{n_{y,y'}(D)}{n_{\cdot,y'}(D)}$$

$$\hat{\tau}_{x,y} = \frac{n_{x,y}(D)}{n_{\cdot,y}(D)}, \text{ where:}$$

$n_{y,y'}(D)$ = number of times state y follows state y' in D

$n_{\cdot,y'}(D) = \sum_{y \in \mathcal{Y}} n_{y,y'}(D)$ = number of times anything follows state y'

$n_{x,y}(D)$ = number of times observation x is generated from y in D

Estimating HMMs from observations alone

- Training data $D = \mathbf{x} = (x_1, \dots, x_n)$, i.e., observations but not the states
 - ▶ E.g., *They neither liked nor disliked the Old Man*
- *Expectation Maximization algorithm* for estimating HMMs from observations alone:

Initialise $\sigma^{(0)}, \tau^{(0)}$ somehow (e.g., randomly)

For iteration $t = 0, 1, 2, \dots$ until converged:

E-step: Calculate $\mathbb{E}_{\sigma^{(t)}, \tau^{(t)}}[n_{y,y'}]$ and $\mathbb{E}_{\sigma^{(t)}, \tau^{(t)}}[n_{x,y}]$

M-step: Calculate $\sigma^{(t+1)}$ and $\tau^{(t+1)}$

$$\sigma_{y,y'}^{(t+1)} = \frac{\mathbb{E}_{\sigma^{(t)}, \tau^{(t)}}[n_{y,y'}]}{\mathbb{E}_{\sigma^{(t)}, \tau^{(t)}}[n_{\cdot,y'}]}$$
$$\tau_{x,y}^{(t+1)} = \frac{\mathbb{E}_{\sigma^{(t)}, \tau^{(t)}}[n_{x,y}]}{\mathbb{E}_{\sigma^{(t)}, \tau^{(t)}}[n_{\cdot,y}]}$$

Calculating expected counts by enumeration

- Training data $D = \mathbf{x} = (x_1, \dots, x_n)$ (no state information)
- Given model σ, τ , the expected values are as follows:

$$E[n_{y', y''}] = \sum_{\mathbf{y} \in \mathcal{Y}^n} n_{y', y''}(\mathbf{x}, \mathbf{y}) P_{\sigma, \tau}(\mathbf{y} | \mathbf{x})$$

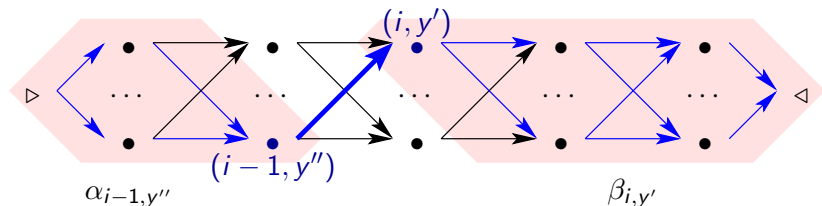
$$E[n_{x', y'}] = \sum_{\mathbf{y} \in \mathcal{Y}^n} n_{x', y'}(\mathbf{x}, \mathbf{y}) P_{\sigma, \tau}(\mathbf{y} | \mathbf{x}), \text{ where:}$$

$$P_{\sigma, \tau}(\mathbf{y} | \mathbf{x}) = \frac{P_{\sigma, \tau}(\mathbf{x}, \mathbf{y})}{\sum_{\mathbf{y}' \in \mathcal{Y}^n} P(\mathbf{x}, \mathbf{y}')}$$

$$P_{\sigma, \tau}(\mathbf{x}, \mathbf{y}) = \left(\prod_{i=1}^n \sigma_{y_i, y_{i-1}} \tau_{x_i, y_i} \right) \sigma_{\triangleleft, y_n}$$

- The *number of state sequences \mathbf{y} grows exponentially with n*
- ⇒ Enumerating all state sequences is infeasible except when \mathbf{x} is very short.

Dynamic programming for computing expected counts



- High level description:
 - ▶ compute *expected number of times each edge in trellis is traversed* $E[N_{(i,y'),(i-1,y'')}]$
 - ▶ sum these over positions $i \in 1, \dots, n+1$ to compute $E[N_{y',y''}]$
- $E[N_{(i,y'),(i-1,y'')}]$ is computed using *forward probability* $\alpha_{i-1,y''}$ and *backward probability* $\beta_{i,y'}$
- *Backward probability* $\beta_{i,y'}$ is sum of weights of all paths from (i, y') to \triangleleft
 - ▶ there is a dynamic programming algorithm for $\beta_{i,y'}$

Further topics on HMMs and sequence modelling

- Higher-order HMMs
 - ▶ The HMMs we investigated here are *first-order*, i.e., the next state y_i is generated from y_{i-1}
 - ▶ In a k -th order HMM, y_i is generated from y_{i-1}, \dots, y_{i-k}
- This theory generalises to *stochastic finite-state transducers* (useful for modelling morphology)

Outline

Overview

n-gram language modelling

Machine translation

Sequence tagging with Hidden Markov Models

Grammars and Syntactic Parsing

PCFGs and beyond for statistical parsing

Why grammars?

Grammars specify a wide range of *sets of structured objects*

- especially useful for describing human languages
- applications in vision, computational biology, etc

There is a *hierarchy* of kinds of grammars

- if a language can be specified by a grammar low in the hierarchy, then it can be specified by a grammar higher in the hierarchy
- the location of a grammar in this hierarchy determines its computational properties

There are generic algorithms for *computing with* and *estimating* (learning) each kind of grammar

- no need to devise new models and corresponding algorithms for each new set of structures

Preview of (P)CFG material

- Why are context-free grammars called “context-free”?
- Context-free grammars (CFG) derivations and parse trees
- Probabilistic CFGs (PCFGs) define probability distributions over derivations/trees
- The number of derivations often grows exponentially with sentence length
- Even so, we can compute the sum/max of probabilities of all trees in cubic time
- It's easy to estimate PCFGs from a treebank (a sequence of trees)
- The EM algorithm can estimate PCFGs from a corpus of strings

Formal languages

T is a finite set of *terminal symbols*, the vocabulary of the language

- E.g., $T = \{\text{likes, Sam, Sasha, thinks}\}$

A *string* is a *finite sequence* of elements of T

- E.g., Sam thinks Sam likes Sasha

T^* is the set of all strings (including the *empty string* ϵ)

T^+ is the set of all non-empty strings

A (formal) *language* is a set of strings (a subset of T^*)

- E.g., $L = \{\text{Sam, Sam thinks, Sasha thinks, ...}\}$

A *probabilistic language* is a probability distribution over a language

Rewrite grammars

A rewrite grammar $G = (T, N, S, R)$ consists of

T , a finite set of *terminal symbols*

N , a finite set of *nonterminal symbols* disjoint from T

$S \in N$ is the *start symbol*, and

R is a finite subset of $N^+ \times (N \cup T)^*$

The members of R are called *rules* or *productions*, and usually written

$\alpha \rightarrow \beta$, where $\alpha \in N^+$ and $\beta \in (N \cup T)^*$

A rewrite grammar defines the *rewrites relation* \Rightarrow , where $\gamma\alpha\delta \Rightarrow \gamma\beta\delta$ iff $\alpha \rightarrow \beta \in R$ and $\gamma, \delta \in (N \cup T)^*$.

A *derivation* of a string $w \in T^*$ is a finite sequence of rewritings

$S \Rightarrow \dots \Rightarrow w$.

\Rightarrow^* is the reflexive transitive closure of \Rightarrow

The language *generated* by G is $\{w : S \Rightarrow^* w, w \in T^*\}$

Example of a rewriting grammar

$G_1 = (T_1, N_1, S, R_1)$, where

$T_1 = \{\text{Al, George, snores}\}$,

$N_1 = \{\text{S, NP, VP}\}$,

$R_1 = \{\text{S} \rightarrow \text{NP VP}, \text{NP} \rightarrow \text{Al}, \text{NP} \rightarrow \text{George}, \text{VP} \rightarrow \text{snores}\}$.

Sample derivations:

$\text{S} \Rightarrow \text{NP VP} \Rightarrow \text{Al VP} \Rightarrow \text{Al snores}$

$\text{S} \Rightarrow \text{NP VP} \Rightarrow \text{George VP} \Rightarrow \text{George snores}$

The Chomsky Hierarchy

Grammars classified by the shape of their productions $\alpha \rightarrow \beta$.

Context-sensitive: $|\alpha| \leq |\beta|$

Context-free: $|\alpha| = 1$

Right-linear: $|\alpha| = 1$ and $\beta \in T^*(N \cup \epsilon)$.

The classes of languages generated by these classes of grammars form a strict hierarchy (ignoring ϵ).

<i>Language class</i>	<i>Recognition complexity</i>
Unrestricted	undecidable
Context-sensitive	exponential time
Context-free	polynomial time
Linear	linear time

Right linear grammars define *finite state languages*, and probabilistic right linear grammars define the same distributions as Hidden Markov Models.

Context-sensitivity in human languages

Some human languages are not context-free (Shieber 1984, Culy 1984). Context-sensitive grammars don't seem useful for describing human languages.

Trees are intuitive descriptions of linguistic structure and are *normal forms* for context-free grammar derivations.

There is an infinite hierarchy of language families (and grammars) between context-free and context-sensitive.

Mildly context-sensitive grammars, such as Tree Adjoining Grammars (Joshi) and Combinatory Categorical Grammar (Steedman) seem useful for natural languages.

Parse trees for context-free grammars

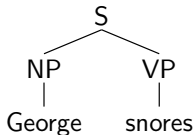
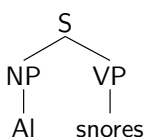
A *parse tree* generated by CFG $G = (T, N, S, R)$ is a finite ordered tree labeled with labels from $N \cup T$, where:

- the root node is labeled S
- for each node n labeled with a nonterminal $A \in N$ there is a rule $A \rightarrow \beta \in R$ and n 's children are labeled β
- each node labeled with a terminal has no children

Ψ_G is the set of all parse trees generated by G .

$\Psi_G(w)$ is the subset of Ψ_G with yield $w \in T^*$.

$$R = \{S \rightarrow NP VP, NP \rightarrow Al, NP \rightarrow George, VP \rightarrow snores\}$$



Example of a CF derivation and parse tree

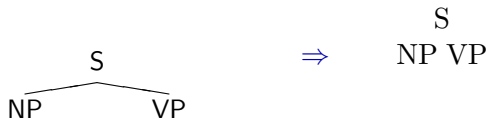
$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow \text{the} & N \rightarrow \text{dog} & V \rightarrow \text{barks} \end{array} \right\}$$

S

S

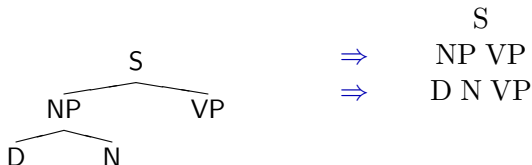
Example of a CF derivation and parse tree

$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow \text{the} & N \rightarrow \text{dog} & V \rightarrow \text{barks} \end{array} \right\}$$



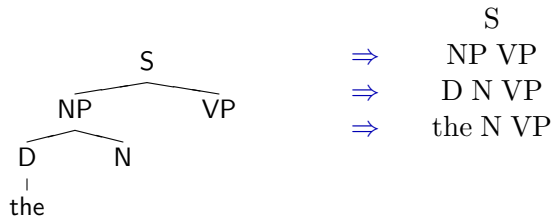
Example of a CF derivation and parse tree

$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow \text{the} & N \rightarrow \text{dog} & V \rightarrow \text{barks} \end{array} \right\}$$



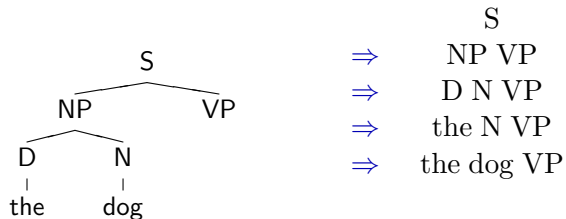
Example of a CF derivation and parse tree

$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow \text{the} & N \rightarrow \text{dog} & V \rightarrow \text{barks} \end{array} \right\}$$



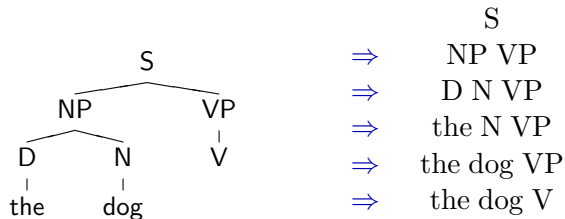
Example of a CF derivation and parse tree

$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow \text{the} & N \rightarrow \text{dog} & V \rightarrow \text{barks} \end{array} \right\}$$



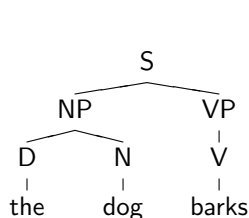
Example of a CF derivation and parse tree

$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow \text{the} & N \rightarrow \text{dog} & V \rightarrow \text{barks} \end{array} \right\}$$



Example of a CF derivation and parse tree

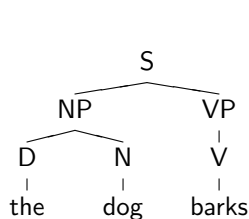
$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow the & N \rightarrow dog & V \rightarrow barks \end{array} \right\}$$



S
⇒ NP VP
⇒ D N VP
⇒ the N VP
⇒ the dog VP
⇒ the dog V
⇒ the dog barks

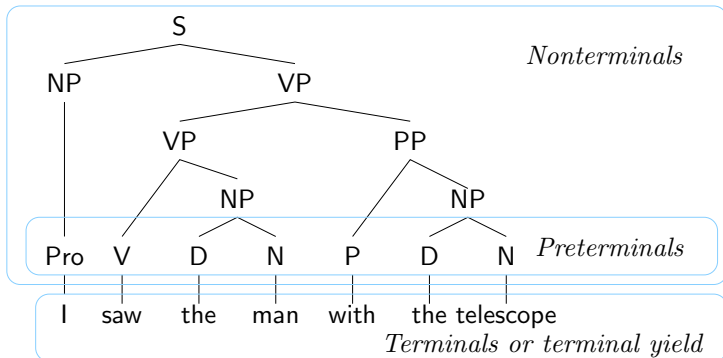
Example of a CF derivation and parse tree

$$R = \left\{ \begin{array}{lll} S \rightarrow NP VP & NP \rightarrow D N & VP \rightarrow V \\ D \rightarrow the & N \rightarrow dog & V \rightarrow barks \end{array} \right\}$$

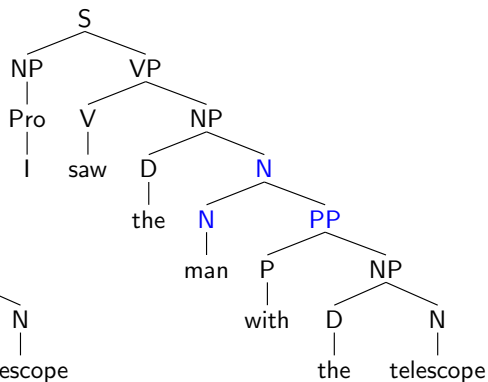
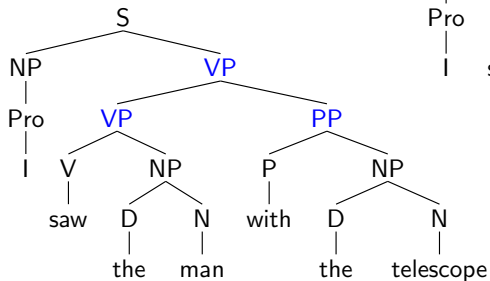


S
⇒ NP VP
⇒ D N VP
⇒ the N VP
⇒ the dog VP
⇒ the dog V
⇒ the dog barks

Trees can depict constituency



CFGs can describe structural ambiguity

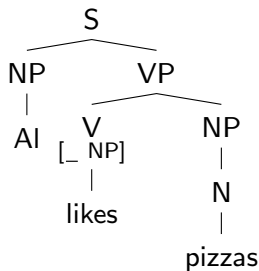
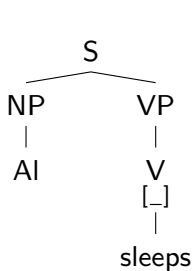


$$R = \{VP \rightarrow V NP, VP \rightarrow VP PP, NP \rightarrow D N, N \rightarrow N PP, \dots\}$$

CFG account of subcategorization

Nouns and verbs differ in the number of *complements* they appear with. We can use a CFG to describe this by splitting or *subcategorizing* the basic categories.

$$R = \left\{ \begin{array}{ll} VP \rightarrow \begin{array}{l} V \\ [-] \end{array} & VP \rightarrow \begin{array}{l} V \\ [- NP] \end{array} NP \\ \begin{array}{l} V \\ [-] \end{array} \rightarrow \text{sleeps} & \begin{array}{l} V \\ [- NP] \end{array} \rightarrow \text{likes} \\ \dots & \dots \end{array} \right\}$$

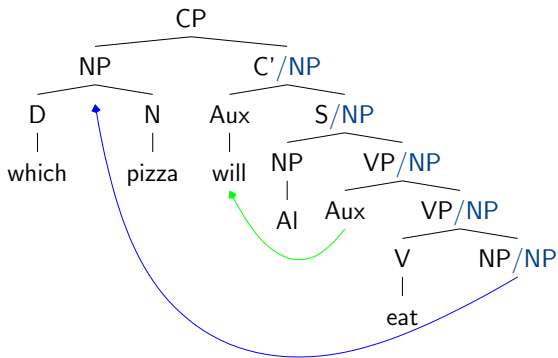
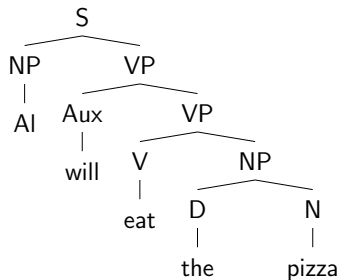


Nonlocal “movement” constructions

“Movement” constructions involve a phrase appearing far from its normal location.

Linguists believed CFGs could not generate them, and posited “movement transformations” to produce them (Chomsky 1957)

But CFGs can generate them via “feature passing” using a conspiracy of rules (Gazdar 1984)



Probabilistic grammars

A probabilistic grammar G defines a probability distribution $P_G(\psi)$ over the parse trees Ψ generated by G , and hence over strings generated by G .

$$P_G(w) = \sum_{\psi \in \Psi_G(w)} P_G(\psi)$$

Standard (non-stochastic) grammars distinguish *grammatical* from *ungrammatical* strings (only the grammatical strings receive parses). Probabilistic grammars can assign non-zero probability to every string, and rely on the probability distribution to distinguish likely from unlikely strings.

Probabilistic context-free grammars

A *Probabilistic Context Free Grammar* (PCFG) consists of (T, N, S, R, p) where:

- (T, N, S, R) is a CFG with no useless productions or nonterminals, and
- p is a vector of *production probabilities*, i.e., a function $R \rightarrow [0, 1]$ that satisfies for each $A \in N$:

$$\sum_{A \rightarrow \beta \in R(A)} p(A \rightarrow \beta) = 1$$

where $R(A) = \{A \rightarrow \alpha : A \rightarrow \alpha \in R\}$.

A production $A \rightarrow \alpha$ is *useless* iff there are no derivations of the form $S \Rightarrow^* \gamma A \delta \Rightarrow \gamma \alpha \delta \Rightarrow^* w$ for any $\gamma, \delta \in (N \cup T)^*$ and $w \in T^*$.

Probability distribution defined by a PCFG

Intuitive interpretation:

- the probability of rewriting nonterminal A to α is $p(A \rightarrow \alpha)$
- the probability of a derivation is the product of probabilities of rules used in the derivation

For each production $A \rightarrow \alpha \in R$, let $f_{A \rightarrow \alpha}(\psi)$ be the number of times $A \rightarrow \alpha$ is used in ψ .

A PCFG G defines a probability distribution P_G on Ψ that is non-zero on $\Psi(G)$:

$$P_G(\psi) = \prod_{r \in R} p(r)^{f_r(\psi)}$$

This distribution is properly normalized if p satisfies suitable constraints.

Example PCFG

1.0 $S \rightarrow NP VP$

0.75 $NP \rightarrow \text{George}$

0.6 $VP \rightarrow \text{barks}$

1.0 $VP \rightarrow V$

0.25 $NP \rightarrow \text{Al}$

0.4 $VP \rightarrow \text{snores}$

$$P \left(\begin{array}{c} S \\ \swarrow \quad \searrow \\ NP \quad VP \\ | \quad | \\ \text{George} \quad V \\ | \\ \text{barks} \end{array} \right) = 0.45$$

$$P \left(\begin{array}{c} S \\ \swarrow \quad \searrow \\ NP \quad VP \\ | \quad | \\ \text{Al} \quad V \\ | \\ \text{snores} \end{array} \right) = 0.1$$

Things we want to compute with PCFGs

Given a PCFG G and a string $w \in T^*$,

- (parsing): the most likely tree for w ,

$$\operatorname{argmax}_{\psi \in \Psi_G(w)} P_G(\psi)$$

- (language modeling): the probability of w ,

$$P_G(w) = \sum_{\psi \in \Psi_G(w)} P_G(\psi)$$

Learning rule probabilities from data:

- (maximum likelihood estimation from visible data): given a corpus of trees $\mathbf{d} = (\psi_1, \dots, \psi_n)$, which rule probabilities p makes \mathbf{d} as likely as possible?
- (maximum likelihood estimation from hidden data): given a corpus of strings $\mathbf{w} = (w_1, \dots, w_n)$, which rule probabilities p makes \mathbf{w} as likely as possible?

Parsing and language modeling

The probability $P_G(\psi)$ of a tree $\psi \in \Psi_G(w)$ is:

$$P_G(\psi) = \prod_{r \in R} p(r)^{f_r(\psi)}$$

Suppose the set of parse trees $\Psi_G(w)$ is finite, and we can enumerate it. Naive parsing/language modeling algorithms for PCFG G and string $w \in T^*$:

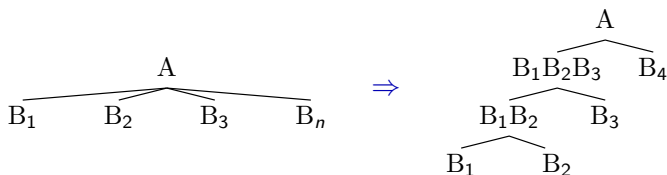
1. Enumerate the set of parse trees $\Psi_G(w)$
2. Compute the probability of each $\psi \in \Psi_G(w)$
3. Argmax/sum as appropriate

Chomsky normal form

A CFG is in *Chomsky Normal Form* (CNF) iff all productions are of the form $A \rightarrow B C$ or $A \rightarrow x$, where $A, B, C \in N$ and $x \in T$.

PCFGs *without epsilon productions* $A \rightarrow \epsilon$ can always be put into CNF.

Key step: *binarize* productions with more than two children by introducing new nonterminals



Substrings and string positions

Let $w = w_1 w_2 \dots w_n$ be a string of length n

A *string position* for w is an integer $i \in 0, \dots, n$ (informally, it identifies the position between words w_{i-1} and w_i)

•	the	•	dog	•	chases	•	cats	•
0		1		2		3		4

A *substring* of w can be specified by beginning and ending string positions $w_{i:j}$ is the substring starting at word $i + 1$ and ending at word j .

$w_{0:4} = \text{the dog chases cats}$

$w_{1:2} = \text{dog}$

$w_{2:4} = \text{chases cats}$

Language modeling using dynamic programming

- **Goal:** To compute $P_G(w) = \sum_{\psi \in \Psi_G(w)} P_G(\psi) = P_G(S \Rightarrow^* w)$
- **Data structure:** A table called a *chart* recording $P_G(A \Rightarrow^* w_{i:k})$ for all $A \in N$ and $0 \leq i < k \leq |w|$
- **Base case:** For all $i = 1, \dots, n$ and $A \rightarrow w_i$, compute:

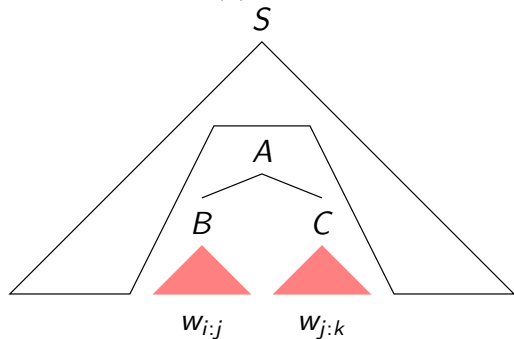
$$P_G(A \Rightarrow^* w_{i-1:i}) = p(A \rightarrow w_i)$$

- **Recursion:** For all $k - i = 2, \dots, n$ and $A \in N$, compute:

$$\begin{aligned} & P_G(A \Rightarrow^* w_{i:k}) \\ &= \sum_{j=i+1}^{k-1} \sum_{A \rightarrow B \ C \in R(A)} p(A \rightarrow B \ C) P_G(B \Rightarrow^* w_{i:j}) P_G(C \Rightarrow^* w_{j:k}) \end{aligned}$$

Dynamic programming recursion

$$P_G(A \Rightarrow^* w_{i:k}) = \sum_{j=i+1}^{k-1} \sum_{A \rightarrow B C \in R(A)} p(A \rightarrow B C) P_G(B \Rightarrow^* w_{i:j}) P_G(C \Rightarrow^* w_{j:k})$$

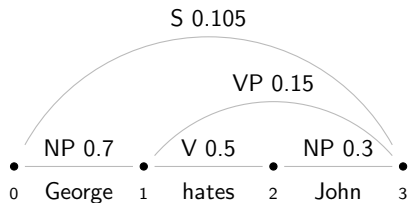


$P_G(A \Rightarrow^* w_{i:k})$ is called the *inside probability* of A spanning $w_{i,k}$.

Example PCFG string probability calculation

$w =$ George hates John

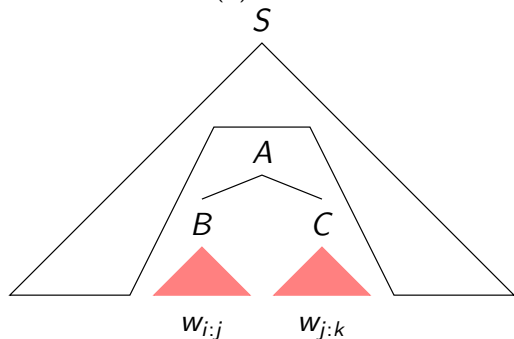
$$R = \left\{ \begin{array}{ll} 1.0 & S \rightarrow NP VP \\ 0.7 & NP \rightarrow \text{George} \\ 0.5 & V \rightarrow \text{likes} \end{array} \right\} \left\{ \begin{array}{ll} 1.0 & VP \rightarrow V NP \\ 0.3 & NP \rightarrow \text{John} \\ 0.5 & V \rightarrow \text{hates} \end{array} \right\}$$



		Right string position		
		1	2	3
Left string position	0	NP 0.7		S 0.105
	1		V 0.5	VP 0.15
	2			NP 0.3

Computational complexity of PCFG parsing

$$P_G(A \Rightarrow^* w_{i,k}) = \sum_{j=i+1}^{k-1} \sum_{A \rightarrow BC \in R(A)} p(A \rightarrow BC) P_G(B \Rightarrow^* w_{i,j}) P_G(C \Rightarrow^* w_{j,k})$$



For each production $r \in R$ and each i, k , we must sum over all intermediate positions $j \Rightarrow O(n^3 |R|)$ time

Estimating (learning) PCFGs from data

Estimating productions and production probabilities from *visible data* (corpus of parse trees) is straight-forward:

- the productions are identified by the local trees in the data
- *Maximum likelihood principle*: select production probabilities in order to make corpus as likely as possible

Estimating production probabilities from *hidden data* (corpus of terminal strings) is much more difficult:

- The *Expectation-Maximization* (EM) algorithm finds probabilities that *locally maximize* likelihood of corpus
- The *Inside-Outside* algorithm runs in cubic time in length of corpus

Estimating PCFGs from visible data

Data: A *treebank* of parse trees $\Psi = \psi_1, \dots, \psi_n$.

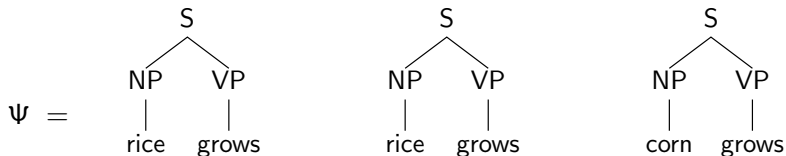
$$L(p) = \prod_{i=1}^n P_G(\psi_i) = \prod_{A \rightarrow \alpha \in R} p(A \rightarrow \alpha)^{f_{A \rightarrow \alpha}(\Psi)}$$

Introduce $|N|$ Lagrange multipliers $c_B, B \in N$ for the constraints $\sum_{B \rightarrow \beta \in R(B)} p(B \rightarrow \beta) = 1$:

$$\frac{\partial \left(L(p) - \sum_{B \in N} c_B \left(\sum_{B \rightarrow \beta \in R(B)} p(B \rightarrow \beta) - 1 \right) \right)}{\partial p(A \rightarrow \alpha)} = \frac{L(p) f_r(\Psi)}{p(A \rightarrow \alpha)} - c_A$$

Setting this to 0, $p(A \rightarrow \alpha) = \frac{f_{A \rightarrow \alpha}(\Psi)}{\sum_{A \rightarrow \alpha' \in R(A)} f_{A \rightarrow \alpha'}(\Psi)}$

Visible PCFG estimation example



Rule

$S \rightarrow NP VP$

$NP \rightarrow \text{rice}$

$NP \rightarrow \text{corn}$

$VP \rightarrow \text{grows}$

Count

3

2

1

3

Rel Freq

1

$2/3$

$1/3$

1

$$P \left(\begin{array}{c} S \\ / \quad \backslash \\ NP \quad VP \\ | \quad | \\ \text{rice} \quad \text{grows} \end{array} \right) = 2/3$$

$$P \left(\begin{array}{c} S \\ / \quad \backslash \\ NP \quad VP \\ | \quad | \\ \text{corn} \quad \text{grows} \end{array} \right) = 1/3$$

Estimating production probabilities from hidden data

Data: A corpus of sentences $\mathbf{w} = w_1, \dots, w_n$.

$$L(\mathbf{w}) = \prod_{i=1}^n P_G(w_i). \quad P_G(w) = \sum_{\psi \in \Psi_G(w)} P_G(\psi).$$

$$\frac{\partial L(\mathbf{w})}{\partial p(A \rightarrow \alpha)} = \frac{L(\mathbf{w}) \sum_{i=1}^n E_G(f_{A \rightarrow \alpha} | w_i)}{p(A \rightarrow \alpha)}$$

Setting this equal to the Lagrange multiplier c_A and imposing the constraint $\sum_{B \rightarrow \beta \in R(B)} p(B \rightarrow \beta) = 1$:

$$p(A \rightarrow \alpha) = \frac{\sum_{i=1}^n E_G(f_{A \rightarrow \alpha} | w_i)}{\sum_{A \rightarrow \alpha' \in R(A)} \sum_{i=1}^n E_G(f_{A \rightarrow \alpha'} | w_i)}$$

This is an iteration of the *expectation maximization* algorithm!

The EM algorithm for PCFGs

Input: a corpus of strings $w = w_1, \dots, w_n$

Guess initial production probabilities $p^{(0)}$

For $t = 1, 2, \dots$ do:

1. Calculate *expected frequency* $\sum_{i=1}^n E_{p^{(t-1)}}(f_{A \rightarrow \alpha} | w_i)$ of each production:

$$E_p(f_{A \rightarrow \alpha} | w) = \sum_{\psi \in \Psi_G(w)} f_{A \rightarrow \alpha}(\psi) P_p(\psi)$$

2. Set $p^{(t)}$ to the *relative expected frequency* of each production

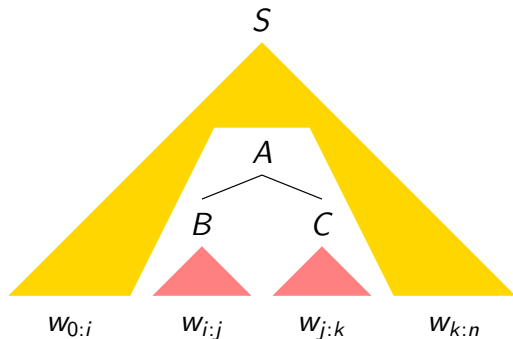
$$p^{(t)}(A \rightarrow \alpha) = \frac{\sum_{i=1}^n E_{p^{(t-1)}}(f_{A \rightarrow \alpha} | w_i)}{\sum_{A \rightarrow \alpha'} \sum_{i=1}^n E_{p^{(t-1)}}(f_{A \rightarrow \alpha'} | w_i)}$$

It is as if $p^{(t)}$ were estimated from a visible corpus Ψ_G in which each tree ψ occurs $\sum_{i=1}^n P_{p^{(t-1)}}(\psi | w_i)$ times.

Dynamic programming for $E_p(f_{A \rightarrow B C} | w)$

$$E_p(f_{A \rightarrow B C} | w) =$$

$$\frac{\sum_{0 \leq i < j < k \leq n} P(S \Rightarrow^* w_{1:i} A w_{k:n}) p(A \rightarrow B C) P(B \Rightarrow^* w_{i:j}) P(C \Rightarrow^* w_{j:k})}{P_G(w)}$$



Calculating “outside probabilities”

Construct a table of “outside probabilities” $P_G(S \Rightarrow^* w_{0:i} A w_{k:n})$ for all $0 \leq i < k \leq n$ and $A \in N$

Recursion from *larger to smaller* substrings in w .

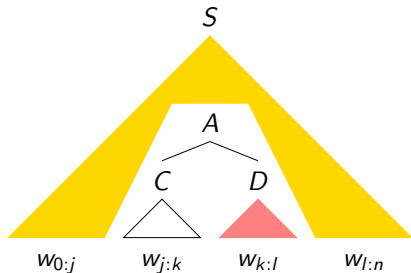
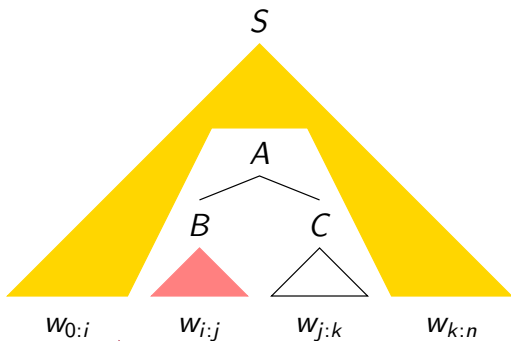
Base case: $P(S \Rightarrow^* w_{0:0} S w_{n:n}) = 1$

Recursion: $P(S \Rightarrow^* w_{0:j} C w_{k:n}) =$

$$\sum_{i=0}^{j-1} \sum_{\substack{A, B \in N \\ A \rightarrow B \ C \in R}} P(S \Rightarrow^* w_{0:i} A w_{k:n}) p(A \rightarrow B \ C) P(B \Rightarrow^* w_{i:j})$$
$$+ \sum_{l=k+1}^n \sum_{\substack{A, D \in N \\ A \rightarrow C \ D \in R}} P(S \Rightarrow^* w_{0:j} A w_{l:n}) p(A \rightarrow C \ D) P(D \Rightarrow^* w_{k:l})$$

Recursion in $P_G(S \Rightarrow^* w_{0:i} A w_{k:n})$

$$\begin{aligned}
 P(S \Rightarrow^* w_{0:j} C w_{k:n}) = & \sum_{i=0}^{j-1} \sum_{\substack{A, B \in N \\ A \rightarrow B C \in R}} P(S \Rightarrow^* w_{0:i} A w_{k:n}) p(A \rightarrow B C) P(B \Rightarrow^* w_{i:j}) \\
 + \sum_{l=k+1}^n \sum_{\substack{A, D \in N \\ A \rightarrow C D \in R}} & P(S \Rightarrow^* w_{0:j} A w_{l:n}) p(A \rightarrow C D) P(D \Rightarrow^* w_{k:l})
 \end{aligned}$$



Outline

Overview

n-gram language modelling

Machine translation

Sequence tagging with Hidden Markov Models

Grammars and Syntactic Parsing

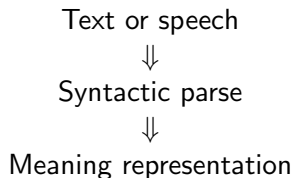
PCFGs and beyond for statistical parsing

What is parsing?

- Syntactic structure depicts:
 - ▶ the way words combine to form phrases and sentences, and
 - ▶ the dependencies between these words and phrases
- A grammar is a finite specification of the syntactic structures of a language
 - ▶ a *probabilistic grammar* defines a probability distribution over syntactic structures
- Parsing is the process of recovering syntactic structures from text or speech

Parsing lies on the path from form to meaning

- “Classical AI”: map text/speech to “meaning representations”
- Might be able to do this if only we knew what “meaning” is
- Problem is figuring out
 - ▶ what information should be in a “meaning representation”, and
 - ▶ what to do with a meaning representation once you have it



Two kinds of uses for probabilistic parsing

“If you build it, they will come . . .”

- Parses as (proxy) meaning representations
 - ▶ machine translation
English text → *English parse* → *French parse* → *French text*
 - ▶ text summarization
 - ▶ information retrieval / question answering
parsing may become more useful with higher bandwidth inputs and lower bandwidth outputs

(should optimize information parses contain for each application)
- Improved language models for “noisy channel” applications
(use parsing model to compute string probability; throw away parse trees)
 - ▶ speech recognition
 - ▶ machine translation

Linguistic grammar-based approaches to parsing

- Division of labour:
 - ▶ Computational linguists
 - design a formalism for stating grammars (e.g., Lexical-functional grammar (LFG), Head-driven phrase-structure grammar (HPSG), Combinatory Categorical Grammar (CCG))
 - design algorithms that find parses generated by *any grammar* written in formalism
 - ▶ Linguists write grammar fragments in formalism that describe interesting constructions
- Central questions:
 - ▶ how complicated does the formalism for stating grammars need to be?
 - ▶ how much computational work is required to *parse*, i.e., find the analysis of a sentence

The ambiguity vs. coverage vs. detail trilemma

- *Ambiguity* grows exponentially with length of sentence
 - Put the rice in a box on the table in the kitchen . . .*
 - ▶ world-knowledge and pragmatics might resolve some ambiguities
 - ▶ many syntactic ambiguities don't correspond to differences in meaning
- *Coverage*: no complete grammars of English or any other language
 - ▶ as you increase coverage, ambiguity usually increases as well
- *Detail*: calling in the linguists often makes matters worse
 - ▶ E.g., systematically distinguishing count and mass nouns adds more ambiguity than it resolves

Statistical approaches to parsing

- Goal: find *most probable analysis* of a string of words
- Approach:
 - ▶ define a probability distribution $P(\text{Tree})$ over *all possible analyses of all possible strings*
 - ▶ from this, we can compute:
 - probability of a tree given words $P(\text{Tree} \mid \text{Words})$
 - probability that a sequence of words is a sentence $P(\text{Words})$ (*language model*)
 - probability that a sequence of words is a *prefix* of a sentence
- Doesn't decide whether an analysis Tree or a string Words is grammatical
 - ▶ usually assign every possible Tree a positive probability
 - ⇒ guarantees *every string gets an analysis*
 - ▶ but these probabilities can *vary enormously*

A new division of labour

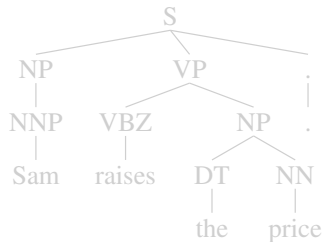
- Why has the statistical approach been so successful?
- Factor the parsing task into two steps:
 - ▶ linguists design (and annotate) the corpora
 - ▶ computer scientists design machine-learning algorithms that generalise from the corpora
- Each group needs only minimal understanding of the other's task

Conventional grammars are closed-world

- The *closed world assumption* for grammars
 - ▶ Rules and lexical entries define set of grammatical structures
 - ▶ Everything not grammatical is ungrammatical
- “Parsing as deduction”: parsing is the process of proving the grammaticality of a sentence
- But: goal is *understanding* what the speaker’s trying to say; not determining whether the sentence is grammatical
- Ideal parser should be *open-world*
 - ▶ even ungrammatical sentences are interpretable
E.g., *man bites dog* \neq *dog bites man*
 - ▶ words and constructions we recognize provide information about sentence’s meaning
 - ▶ unknown words or phrases do not cause interpretation to fail
 - parsing and acquisition are two aspects of same process
- *Statistical treebank parsers are open-world*
 - ▶ every possible tree receives positive probability

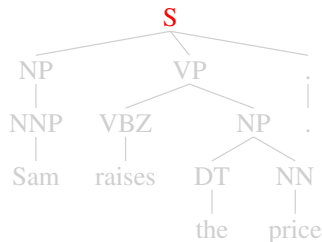
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



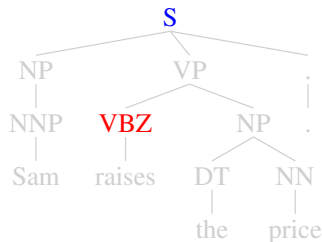
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



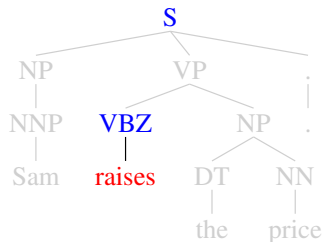
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



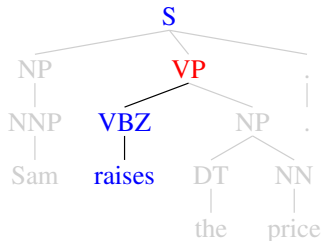
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



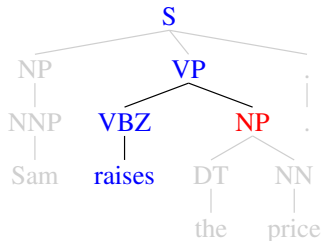
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



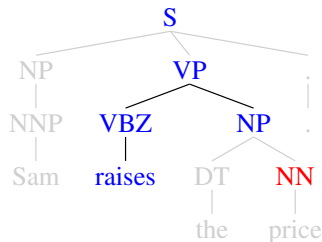
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



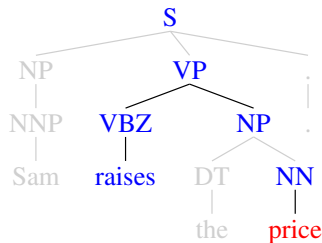
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



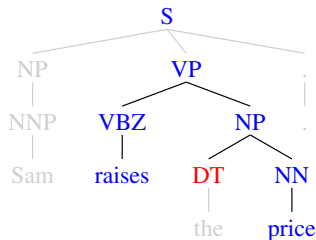
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



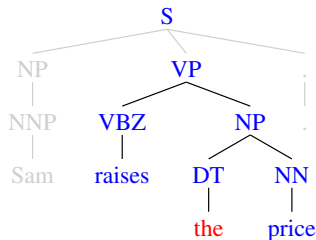
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



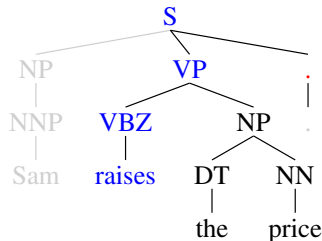
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



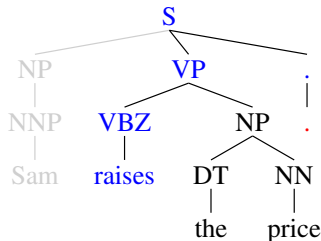
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



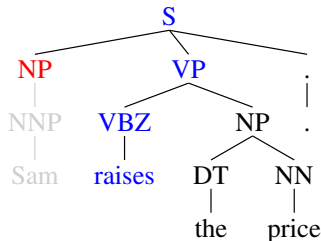
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



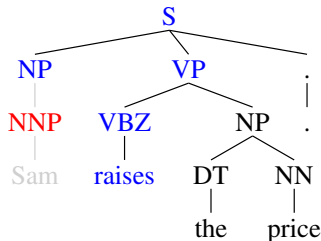
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



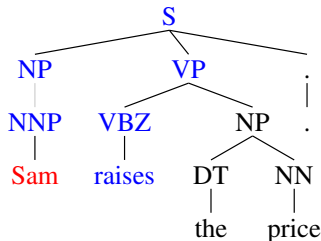
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



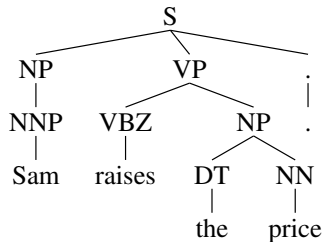
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



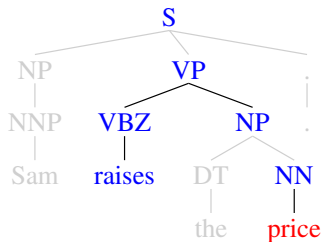
A “history-based” generative parsing model

- Generates nodes of tree in (roughly) top-down order
- *Predict each node from nodes already generated:*
 - ▶ governor and its maximal projection
 - ▶ governor's governor and its maximal projection



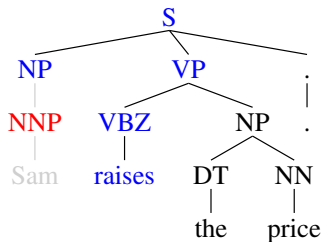
Estimating grammar probabilities from treebanks

- The statistical parser needs to know the conditional probability of generating node type at each location in the tree
- A *treebank* is a corpus in which each sentence is labelled with its analysis tree
- The Penn WSJ treebank contains parses for $\sim 40,000$ sentences (1.2 million words)
- The conditional probabilities can be estimated from a treebank by *counting how often each node type appears in each context*
- There are methods for learning these probabilities from strings alone, but the resulting grammars are not very good



Capturing vs. covering linguistic generalizations

- *Capturing a generalization*: grammar accurately describes phenomenon at appropriate level, e.g., subject-verb agreement via PERSON and NUMBER features
- *Covering a generalization*: model covers common cases of a generalization, perhaps indirectly. E.g., head-to-head POS dependencies
- An “engineering” parser only needs to cover generalizations
- Even so, feature design requires *linguistic insight*
 - ▶ basic linguistic insights are often most useful

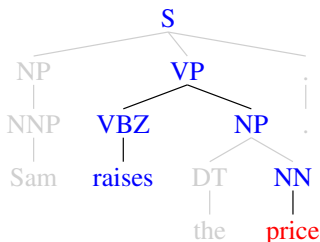


Parsing in the late 1990s


- Parsers for hand-written grammars (LFG, HPSG, etc)
 - ▶ linguistically rich, detailed representations
 - ▶ uneven / poor coverage of English
 - ▶ even simple sentences are highly ambiguous
 - ▶ only ad hoc treatment of preferences
 - ▶ could not be learnt from data
- Generative probabilistic parsers
 - ▶ systematic treatment of preferences
 - ▶ *learnt from treebank corpora*
 - ▶ simple constituent structure representations
 - ▶ wide (if superficial) coverage of English
- *Could the two approaches be combined?*

Generative statistical parsers

- “History-based” generative statistical parsers (Bikel, Charniak, Collins) generate each new node in parse conditioned on the structure already generated:

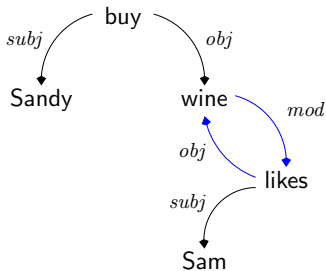


$$P(\text{price}|\text{NN}, \text{NP}, \text{raises}, \text{VBZ}, \text{VP}, \text{S})$$

- Assume each node is independent of all existing structure except for nodes explicitly conditioned on
- ⇒ $P(t)$ is product of node probabilities
- ⇒ simple “relative frequency” estimators (smoothing is essential )

Reentrant dependencies cause problems

- Linguistic structures exhibit *cyclic dependencies*
 - ▶ Linguistically-motivated grammars typically include many kinds of these
 - Cannot generate such parses as sequence of independent steps (naive approach “double counts”)
- ⇒ Estimating parameters of such grammars is computationally much harder



“Sandy bought the wine
Sam likes”

Generalising beyond generative models

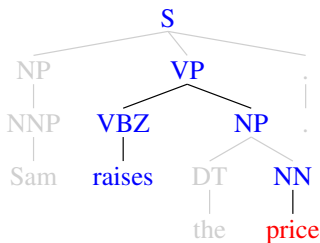
- Mathematically straight-forward to define models in which parse is is not generated as a sequence of independent decisions
 - ▶ Maximum Entropy, log-linear, exponential, Gibbs, ...

$$P(t) = \frac{1}{Z} \exp \sum_{j=1}^m w_j f_j(t)$$

- Once we no longer require factors to be independent
 - ▶ parses t need not be trees
 - ▶ features f_j can be *any computable function* of parses
e.g., one feature for each possible kind of dependency
- But simple “relative frequency” estimators for feature weights w no longer work; maximum likelihood estimation is very hard

Abney (1997)

Log-linear models are more expressive



- Define features $f_{x,c}$ for all node labels x and contexts c
 $f_{(\text{price}, \text{NN}, \text{NP}, \text{raises}, \text{VBZ}, \text{VP}, \text{S})}(t)$ is the number of times
(price, NN, NP, raises, VBZ, VP, S) appears in parse t
- Let weight $w_{x,c} = \log P(x|c)$
 $w_{(\text{price}, \text{NN}, \text{NP}, \text{raises}, \text{VBZ}, \text{VP}, \text{S})} = \log P(\text{price}|\text{NN}, \text{NP}, \text{raises}, \text{VBZ}, \text{VP}, \text{S})$
- Generative and log-linear models define same prob. distribution
- Easy to add additional features to log-linear model, but difficult to add additional features to the generative model

Why is maximum likelihood estimation hard?

- Given treebank training corpus $D = (t_1^*, \dots, t_n^*)$, *maximum likelihood estimator* adjusts feature weights w to maximize $P(D) = \prod_i P(t_i^*)$

$$P(t) = \frac{1}{Z} \exp \sum_{j=1}^m w_j f_j(t)$$

$$Z = \sum_{t' \in \mathcal{T}} \exp \sum_{j=1}^m w_j f_j(t')$$

where \mathcal{T} is set of *all possible trees for all possible strings*

- ML estimator adjusts w to make trees in D more likely than any other trees
- Features form a tree of independent generative steps $\Rightarrow Z = 1$
 \Rightarrow estimation easy
- Features have complex dependencies $\Rightarrow Z \neq 1 \Rightarrow$ estimation hard

Conditional maximum likelihood estimation

- *Most likely parse* $\hat{t}(s)$ for string s only depends on $P(t|s)$
- ⇒ only need to estimate $P(t|s)$

$$\hat{t}(s) = \operatorname{argmax}_{t \in \mathcal{T}(s)} P(t|s), \text{ where } \mathcal{T}(s) = \text{set of possible trees for } s$$

$$P(t|s) = \frac{1}{Z(s)} \exp \sum_j w_j f_j(t); \quad Z(s) = \sum_{t \in \mathcal{T}(s)} \exp \sum_j w_j f_j(t)$$

- Partition functions $Z(s_i)$ “only” require summing over $\mathcal{T}(s_i)$, i.e., all parses for each string s_i in training data D
- *Conditional* maximum likelihood selects w to optimize conditional probability of training data $D = ((s_1, t_1^*), \dots, (s_n, t_n^*))$

$$\hat{w} = \operatorname{argmax}_w \prod_{i=1}^n P(t_i^* | s_i)$$

Johnson, Geman, Canon, Chi and Riezler (1999)

Conditional estimation

s_i	$f(t_i^*)$	feature vectors of other parses for s_i
sentence 1	(1, 3, 2)	(2, 2, 3) (3, 1, 5) (2, 6, 3)
sentence 2	(7, 2, 1)	(2, 5, 5)
sentence 3	(2, 4, 2)	(1, 1, 7) (7, 2, 1)
...

- Treebank training data D provides correct parse t_i^* for sentence s_i
- Parser produces all possible parse trees for each sentence s
- Feature extractor extracts feature vectors $(f_1(t), \dots, f_m(t))$ for each parse tree t
- Estimator selects feature weights $w = (w_1, \dots, w_m)$ to make each t_i^* score as high as possible relative to other parses for s_i

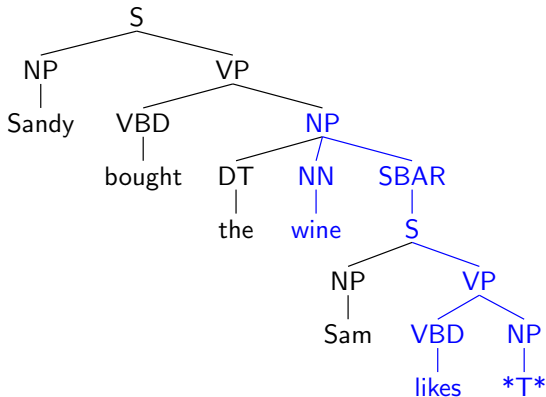
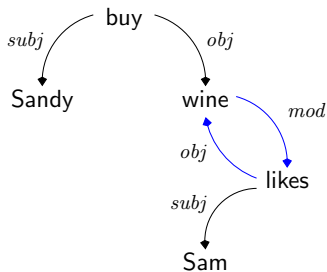
Conditional vs joint estimation

$$P(t, s) = P(t|s)P(s)$$

- MLE maximizes probability of training trees t^* *and strings* s
 - ▶ Can be used for *language modeling* $P(s) = \sum_{t \in \mathcal{T}(s)} P(t, s)$
 - ▶ Extends to unsupervised estimation via EM
- Conditional MLE maximizes probability of t^* *given strings* s
 - ▶ estimates exactly what we need for parsing
 - ▶ uses less information from the data (ignores $P(s)$)
 - ▶ ignores unambiguous sentences (i.e., $|\mathcal{T}(s)| = 1$)
- Joint estimation should be better (lower variance) if model correctly relates $P(t|s)$ and $P(s)$
- Conditional estimation should be better if model incorrectly relates $P(t|s)$ and $P(s)$

Details of syntactic representation don't matter

- Log-linear models make decisions based only on *feature vectors* $f(t)$
- ⇒ Representation t doesn't matter as long as t is “rich enough” to define $f_j(t)$



Conditional estimation for parse rescoring

- Easy to over-fit training data with large number of features
- ⇒ Regularize by adding a penalty term to log likelihood
 - ▶ L1 penalty term ⇒ sparse feature weight vector
 - ▶ L2 penalty term (Gaussian prior) seems best
- 50-best parses $\mathcal{T}(s)$ may not include true parse $t^*(s)$
- ⇒ Train rescorer to prefer parse in $\mathcal{T}_c(s)$ closest to $t^*(s)$
 - Often several parses from 50-best list are equally close to true parse
- ⇒ EM-inspired (non-convex) loss function
 - Direct numerical optimization with L-BFGS (modified for L1 regularizer) produces best results

Riezler, King, Kaplan, Crouch, Maxwell and Johnson (2002),
Goodman (2004), Andrew and Gao (2007)

Features for rescoring parses

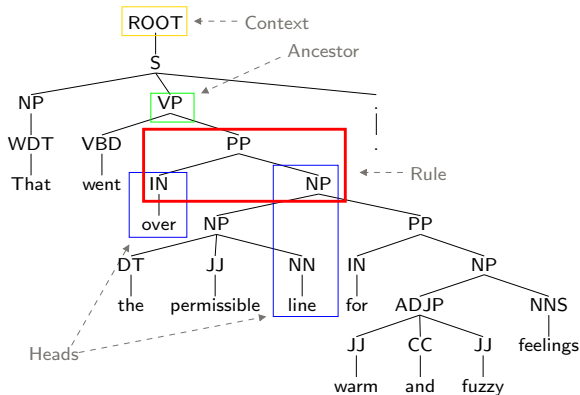
- Parse rescorer's features can be *any computable function* of parse
- Choice of features is the most important and least understood aspect of the parser
 - ▶ feature design has a much greater impact on performance than the learning algorithm
- Features can be based on a linguistic theory (or more than one)
- ... but need not be
 - ▶ “shot-gun” or “hail Mary” features very useful
- Feature selection: a feature's values on $T^*(s)$ and $\mathcal{T}(s) \setminus T^*(s)$ must differ on at least 5 sentences $s \in D$
- The *Charniak parser's log probability* combines all of the generative parser's conditional distributions into a single rescorer feature \Rightarrow rescoring should never hurt

Lexicalized and parent-annotated rules

Lexicalization associates each constituent with its head

Ancestor annotation provides a little “vertical context”

Context annotation indicates constructions that only occur in main clause (c.f., Emonds)



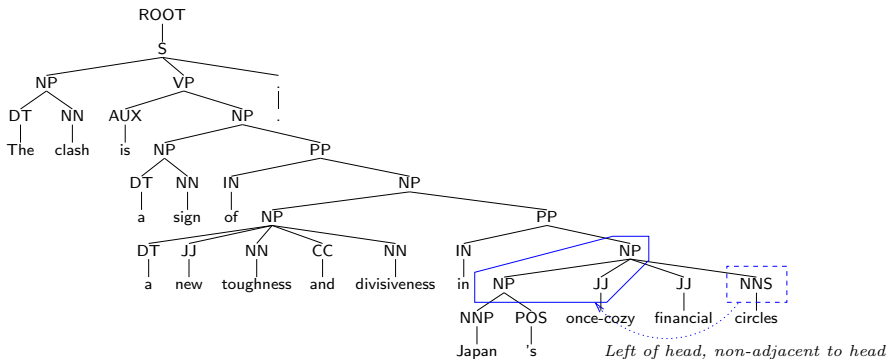
n-gram rule features generalize rules

Collects *adjacent constituents* in a local tree

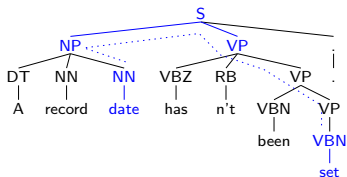
Also includes *relationship to head* (e.g., adjacent? left or right?)

Parameterized by *ancestor-annotation*, *lexicalization* and *head-type*

There are 5,143 unlexicalized rule bigram features and 43,480 lexicalized rule bigram features



Bihead dependency features

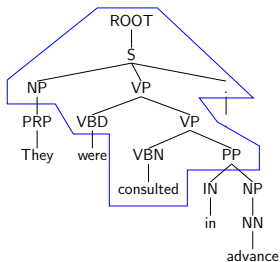


(S (NP (NP (NN date))) (VP (VP (VBZ has) (RB n't) (VP (VBN been) (VP (VBN set)))))

- Bihead dependency features approximate linguistic function-argument dependencies
- Computed for lexical (\approx semantic) and functional (\approx syntactic) heads
- One feature for each head-to-head dependency found in training corpus (70,000 features in all)

Head trees record all dependencies

- Head trees consist of a (lexical) head, all of its projections and (optionally) all of the siblings of these nodes
- correspond roughly to TAG elementary trees
- parameterized by *head type*, *number of sister nodes* and *lexicalization*

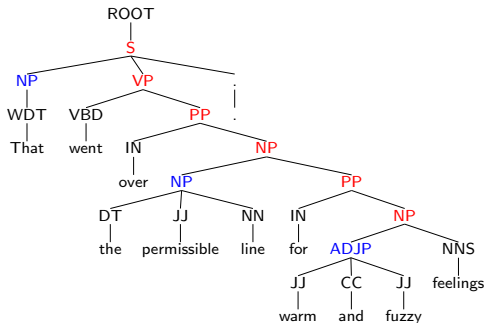


Rightmost branch bias

The RightBranch feature's value is the number of nodes on the right-most branch (ignoring punctuation) (c.f., Charniak 00)

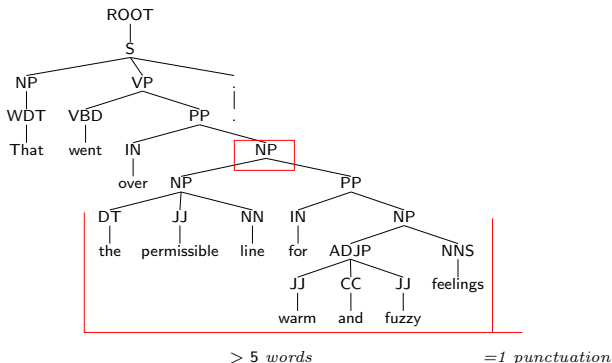
Reflects the tendency toward right branching in English

Only 2 different features, but very useful in final model!



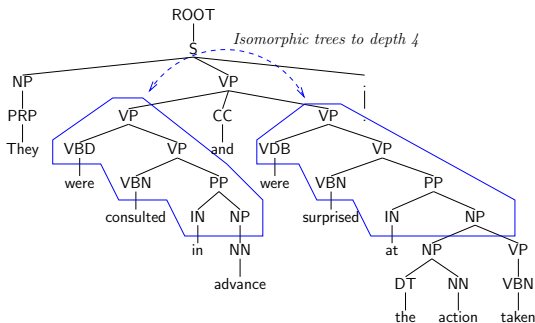
Constituent Heaviness and location

- Heaviness measures the constituent's category, its (binned) size and (binned) closeness to the end of the sentence



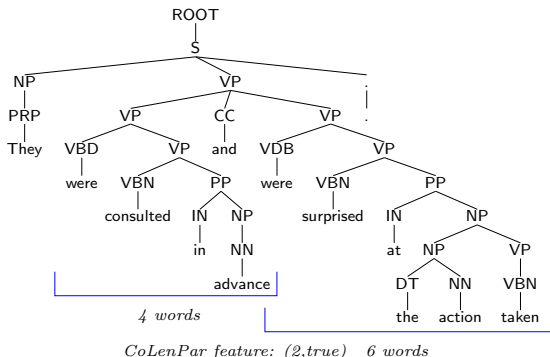
Coordination parallelism (1)

- A CoPar feature indicates the depth to which adjacent conjuncts are parallel



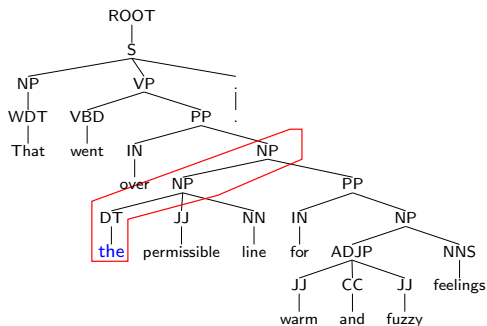
Coordination parallelism (2)

- The CoLenPar feature indicates the difference in length in adjacent conjuncts and whether this pair contains the last conjunct.

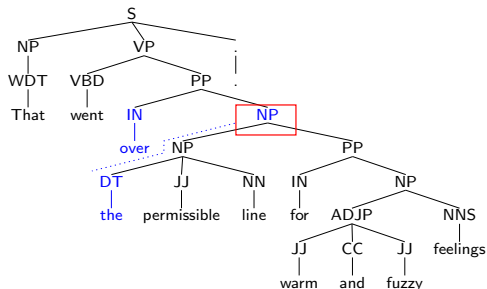


Word

- A Word feature is a word plus n of its parents (c.f., Klein and Manning's non-lexicalized PCFG)
- A WProj feature is a word plus all of its (maximal projection) parents, up to its governor's maximal projection



Constituent “edge neighbour” features

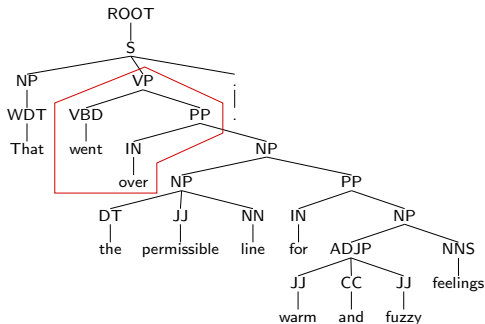


(IN over) (NP (DT the ...))

- Edge features are a kind of bigram context for constituents
- Would be difficult to incorporate into a generative parser

Tree n -gram

- A tree n -gram feature is a tree fragment that connect sequences of adjacent n words, for $n = 2, 3, 4$ (c.f. Bod's DOP models)
- lexicalized and non-lexicalized variants
- There are 62,487 tree n -gram features



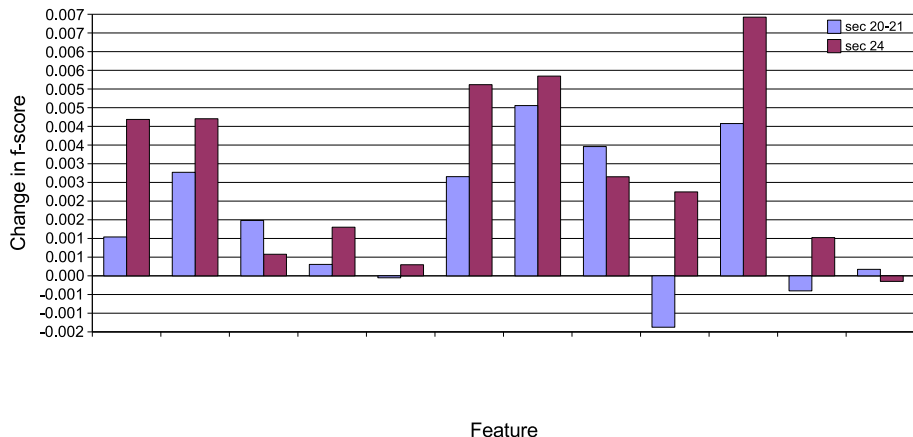
Experimental setup

- Feature tuning experiments done using Collins' split: sections 2-19 as train, 20-21 as dev and 24 as test
- $\mathcal{T}_c(s)$ computed using Charniak 50-best parser
- Features which vary on less than 5 sentences pruned
- Optimization performed using LMVM optimizer from Petsc/TAO optimization package or Averaged Perceptron
- Regularizer constant c adjusted to maximize f-score on dev

Evaluating features

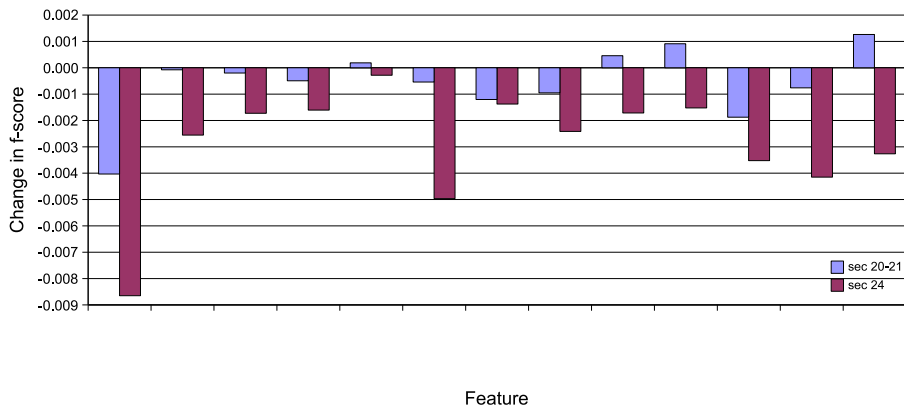
- The feature weights are not that indicative of how important a feature is
- The MaxEnt ranker with regularizer tuning takes approx 1 day to train
- The *averaged perceptron* algorithm takes approximately 2 minutes
 - ▶ used in experiments comparing different sets of features
 - ▶ Used to compare models with the following features:
**NLogP Rule NGram Word WProj RightBranch Heavy NGramTree
HeadTree Heads Neighbours CoPar CoLenPar**

Adding one feature class



- Averaged perceptron baseline with only base parser log prob feature
 - ▶ section 20–21 f-score = 0.894913
 - ▶ section 24 f-score = 0.889901

Subtracting one feature class



- Averaged perceptron baseline with all features
 - ▶ section 20–21 f-score = 0.906806
 - ▶ section 24 f-score = 0.902782

Comparing estimators

- Training on sections 2–19, regularizer tuned on 20–21, evaluate on 24

Estimator	# features	sec 20-21	sec 24
MaxEnt model, $p = 2$	670,688	0.9085	0.9037
MaxEnt model, $p = 1$	14,549	0.9078	0.9024
averaged perceptron	523,374	0.9068	0.9028
expected f-score	670,688	0.9084	0.9029

- None of the differences are significant
- Because the exponential model with $p = 2$ was the first model I tested new features on, they may be biased to work well with it.

Further directions in syntactic parsing

- *Self-training*: semi-supervised learning using the parser's trees as input for training
- *Automatic domain adaptation*: selecting an optimal combination of treebanks for parser training based on properties of the corpus
- *Parsing speech recognizer output*: detecting and partially correcting disfluencies; using prosody in place of punctuation

Interested in statistical models for computational linguistics?

We're recruiting *PhD students* and *post-docs*.

Contact Mark.Johnson@mq.edu.au for more information.

