CSCI 4152/6509 — Natural Language Processing

4-Nov-2009

Lecture 22: Probabilistic Parsing

Room: FASS 2176
Time: 11:35 – 12:25

Previous Lecture

- The most probable completion in Bayesian Networks;
- HMM as a BayesianNetwork,
- HMM completion example;
- Probabilistic Context Free Grammars:
  - motivation,
  - PCFG as a probabilistic model

13.2 PCFG as a Probabilistic Model

To transform a CFG into a probabilistic model we model derivations as stochastic process in a generative way. For example, the left-most derivation corresponding to the first parse tree described above is:

$$\begin{aligned} \text{S} &\Rightarrow \text{NP VP} \Rightarrow \text{N VP} \Rightarrow \text{time VP} \Rightarrow \text{time V PP} \Rightarrow \text{time flies PP} \Rightarrow \text{time flies P NP} \\ &\Rightarrow \text{time flies like NP} \Rightarrow \text{time flies like D N} \Rightarrow \text{time flies like an N} \Rightarrow \text{time flies like an arrow} \end{aligned}$$

At each step of the derivation, given a non-terminal that needs to be re-written, we usually have several options, corresponding to several rules that have this non-terminal on the left-hand side.
Hence, we calculate the probability of the tree by multiplying probabilities of all rules occurring in the tree:

$$\begin{aligned} \mathrm{P}(\text{first tree}) &= \mathrm{P}(\text{N} \rightarrow \text{time})\mathrm{P}(\text{V} \rightarrow \text{flies})\mathrm{P}(\text{P} \rightarrow \text{like})\mathrm{P}(\text{D} \rightarrow \text{an}) \\ &\quad \mathrm{P}(\text{N} \rightarrow \text{arrow})\mathrm{P}(\text{NP} \rightarrow \text{N})\mathrm{P}(\text{NP} \rightarrow \text{D N}) \ldots \mathrm{P}(\text{S} \rightarrow \text{NP VP}) \end{aligned}$$

If we assign the following probabilities to the rules:

S	→	NP VP	/1	VP	→	V NP	/.5	N	→	time	/.5
NP	→	N	/.4	VP	→	V PP	/.5	N	→	arrow	/.3
NP	→	N N	/.2	PP	→	P NP	/1	N	→	flies	/.2
NP	→	D N	/.4					D	→	an	/1
V	→	like	/.3								
V	→	flies	/.7								
P	→	like	/1								

then the probability of the first tree is 0.0084, and the probability of the second tree is 0.00036. We can conclude that the first tree is more likely, which should correspond to our intuition.
The probability assigned to a rule $N \rightarrow \alpha$ is the probability $\mathrm{P}(N \rightarrow \alpha | N)$, so if $N \rightarrow \alpha_1$, $N \rightarrow \alpha_2, \ldots, N \rightarrow \alpha_n$ are all rules with the nonterminal N on its left hand side, then

$$\sum_{i=1}^{n} \mathrm{P}(N \rightarrow \alpha_i) = 1$$

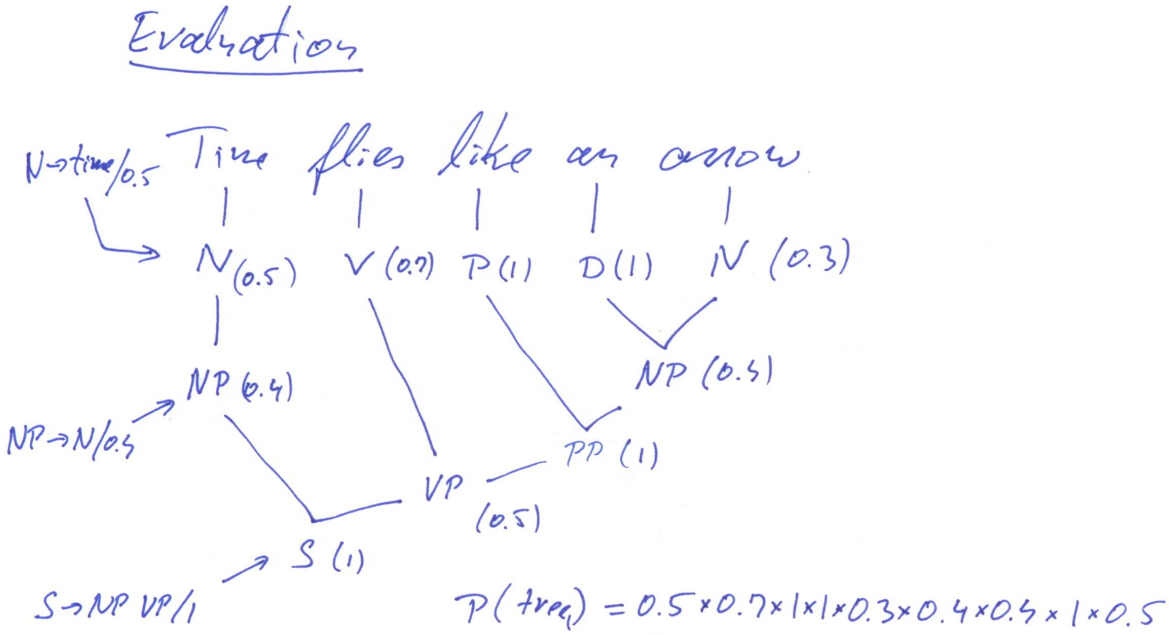
These probabilities are easily learned from a set of parse trees, usually called parse treebank, by counting the number of occurrences of distinct rules.

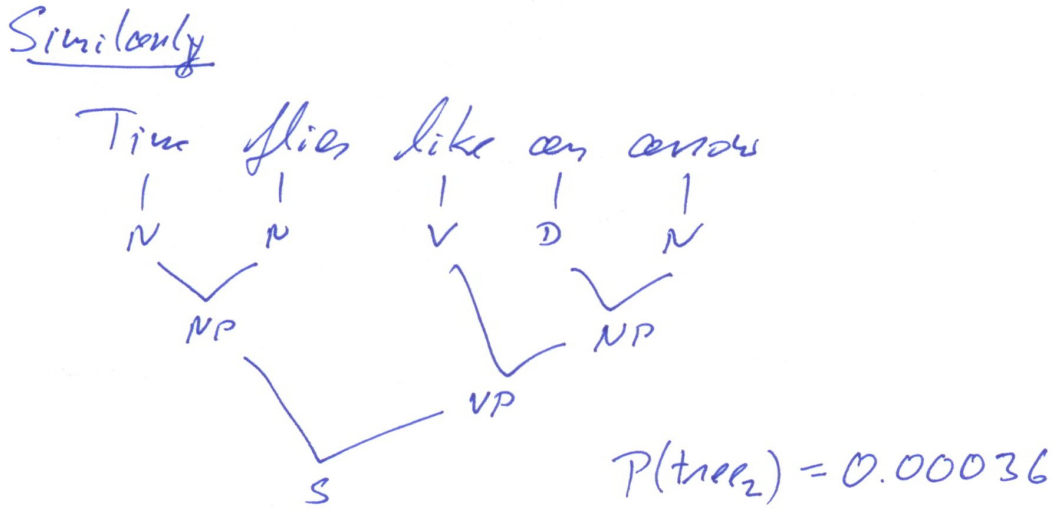
This model is a language model, since the sum of probabilities of all sentences in the language is 1. Actually, in order to be a language model, we also require that the grammar is *proper,* i.e., that all infinite trees have probability 0, which is not always the case. We will not go into further details regarding this question here, except noting that it has been proved that any PCFG with probabilities induced from a treebank is proper.

Computational Tasks for PCFG Model

Evaluation

Given a tree t, what is $\mathrm{P}(t)$? As seen in the example, we simply multiply probabilities associated with each node of the tree.





Generation

We can generate (sample) sentences by starting with the initial nonterminal S, and by rewriting it using a rule $S \rightarrow \alpha$ according to the probabilities assigned to the rules that have S on their left hand side. We obtain a *sentential form*—a string of terminals and nonterminals. Take the first nonterminal in this form, and rewrite it in the same way; and so on. The loop stops when there are no nonterminals, i.e., when we obtain a sample sentence. An interesting question is whether this process stops. If the grammar is *proper*, it stops with probability 1. If the grammar is not proper, we might easily be trapped in an infinite derivation.

Generation (sampling)

S => NP VP => N VP => flies VP => ...

S → NP VP / 1

NP → N / 0.4
NP → N N / 0.2
NP → D N / 0.4

N → time / 0.5
N → arrow / 0.3
N → flies / 0.2

- choose rule randomly according to the given distribution

Question: Is the process going to stop?
A: Stops with probability 1 if the grammar is proper.

Good News: A grammar learned from a corpus is always proper.

Learning

An direct approach to learning from a completely given parse treebank is counting the number of rule occurrences.

Inference

Marginalization: $\mathrm{P}(\text{sentence}) = ?$

Conditioning: $\mathrm{P}(\text{tree}|\text{sentence}) = ?$

Completion: $\arg\max_{\text{tree}} \mathrm{P}(\text{tree}|\text{sentence})$

13.3 Efficient Inference in PCFG Model

Let us consider the marginalization task:
$\mathrm{P}(\text{sentence}) = ?$

If 'sentence' is the following sequence of words: $w_1 w_2 \dots w_n$, then $\mathrm{P}(\text{sentence})$ is the following conditional probability:

$$\mathrm{P}(\text{sentence}) = \mathrm{P}(w_1 w_2 \dots w_n | S)$$

i.e., it is the probability of generating the sentence given that we start from S, i.e, it is $\mathrm{P}(S \Rightarrow^* w_1 \dots w_n)$.
An obvious way to calculate this marginal probability is to find all parse trees of a sentence and sum their probabilities, i.e:

$$\mathrm{P}(\text{sentence}) = \sum_{t \in T} \mathrm{P}(t),$$

where T is the set of all parse trees of the sentence 'sentence'. However, this may be very inefficient. We also need a way to find all parse trees.
As an example illustrating that the above direct approach may lead to an exponential algorithm, consider a CFG with only two rules $S \Rightarrow S\ S$ and $S \Rightarrow a$. The sentences a^n have as many parse trees as there are binary trees with n leaves, which is a well-known Catalan number, $\approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$ as $n \to \infty$.
An algorithm for efficient marginalization can be derived from the well-known efficient parsing algorithm known as CYK (Cocke-Younger-Kasami) algorithm. The algorithm has a running-time complexity of $O(n^3)$ for a sentence of length n.
CYK can be applied only to a CNF (Chomsky Normal Form) grammar, so if the grammar is not already in CNF, we would have to convert it to CNF. A Context-Free Grammar is in CNF if all its rules are either of the form $A \to B\,C$, where A, B, and C are nonterminals, or $A \to w$, where A is a nonterminal and w is a terminal. If a CFG is not in CNF, it can be converted into CNF.
Is the following grammar in CNF?

S	→	NP VP	VP	→	V NP	N	→	time	V	→	like
NP	→	N	VP	→	V PP	N	→	arrow	V	→	flies
NP	→	N N	PP	→	P NP	N	→	flies	P	→	like
NP	→	D N				D	→	an			

How about this grammar?

S	→	NP VP	VP	→	V NP	N	→	time	V	→	like
NP	→	time	VP	→	V PP	N	→	arrow	V	→	flies
NP	→	N N	PP	→	P NP	N	→	flies	P	→	like
NP	→	D N				D	→	an			

Note: What if the grammar is not in CNF

There is a standard algorithm for converting arbitrary CFG into CNF. The problem is: How to calculate probabilities of the rules in the new grammar? One way is to sample from the old grammar, and to estimate probabilities in the new grammar by parsing the sample sentences and counting. The probabilities can also be calculated directly, but it not a straightforward task.
Here are the steps needed for conversion of an arbitrary CFG into CNF. This is just a partial sketch of the algorithm: calculating probabilities of the new rules in the first two steps is not trivial and it is not given.

Eliminate empty rules $N \to \epsilon$

Find all "nullable" nonterminals, i.e., terminals N such that $N \Rightarrow^* \epsilon$.
From each rule $A \to X_1 \dots X_n$ create new rules by striking out some nullable nonterminals. This is done for all combination of nonterminals in the rule, except for striking out all $X_1 \dots X_n$ if they are all nullable.
Remove empty rules.
If the start symbol is nullable, add $S \to \epsilon$, and treat that as a special case.

Eliminate unit rules $N \to M$

For any two variables A and B, such that $A \Rightarrow^* B$, for all non-unit rules $B \to \zeta$, we add $A \to \zeta$. Remove unit rules.
All possible derivations $A \Rightarrow^* B$ are easy to find since the empty rules are already eliminated.

Eliminate terminals in rules, except $A \to w$

For each terminal w that appears on the right hand side of some rule with some other symbols, we introduce a new nonterminal N_w, and a rule $N_w \to w$ with probability 1. Then we replace w in all other rules with N_w.

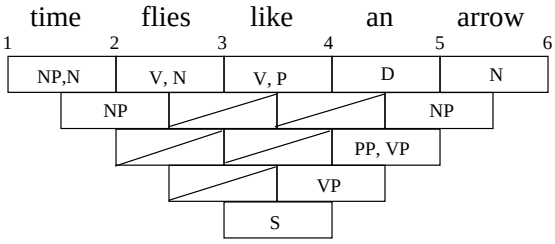
Eliminate rules $A \to B_1 B_2 \dots B_n$ **(**$n > 2$**)**

For each rule $A \to B_1 B_2 \dots B_n$ ($n > 2$), we introduce $n-2$ new nonterminals $X_1, \dots, X_{n-2}$, and replace this rule with the following rules: $A \to B_1 X_1$, $X_1 \to B_2 X_2, \dots X_{n-2} \to B_{n-1} B_n$, and assign the following probabilities to them: $\mathrm{P}(A \to B_1 X_1) = \mathrm{P}(A \to B_1 B_2 \dots B_n)$, $\mathrm{P}(X_1 \to B_2 X_2) = 1, \dots \mathrm{P}(X_{n-2} \to B_{n-1} B_n) = 1$.

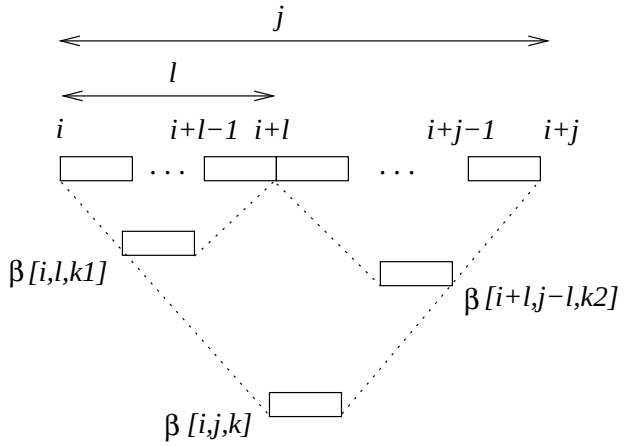
CYK Example

The following grammar in CNF is given:

S	→	NP VP	VP	→	V NP	N	→	time	V	→	like
NP	→	time	VP	→	V PP	N	→	arrow	V	→	flies
NP	→	N N	PP	→	P NP	N	→	flies	P	→	like
NP	→	D N				D	→	an			



Explanation of Index use in CYK



CYK Algorithm

Let all nonterminals be: $N^1, \dots N^m$.

In the standard CYK algorithm, we have a two dimensional table β in which only the entries β_{ij}, $1 \leq i \leq i + j - 1 \leq n$, are used. Each entry β_{ij} contains a set of nonterminals that can produce substring $w_i \dots w_{i+j-1}$ using the grammar rules, i.e., $\beta_{ij} = \{N | N \Rightarrow^* w_i \dots w_{i+j-1}\}$.
If we enumerate all nonterminals: N^1, N^2, …, N^m, then each set of nonterminals β_{ij} can be represented by extending β to be a 3-dimensional table β_{ijk}, in which $\beta_{ijk} = 1$ means that N^k can produce substring $w_i \dots w_{i+j-1}$, and $\beta_{ijk} = 0$ that it cannot.

Algorithm 1 CYK

Require: sentence $= w_1 \dots w_n$, and a CFG in CNF with nonterminals $N^1 \dots N^m$,
N^1 is the start symbol
Ensure: parsed sentence

```
allocate matrix β ∈ {0,1}^{n×n×m} and initialize all entries to 0
for i ← 1 to n do
    for all rules N^k → w_i do
        β[i, 1, k] ← 1
for j ← 2 to n do
    for i ← 1 to n − j + 1 do
        for l ← 1 to j − 1 do
            for all rules N^k → N^{k_1} N^{k_2} do
                β[i, j, k] ← β[i, j, k] OR (β[i, l, k_1] AND β[i + l, j − l, k_2])
return β[1, n, 1]
```