Faculty of Computer Science, Dalhousie University *28-Nov-2023*

CSCI 4152/6509 — Natural Language Processing

Lecture 23: DCG and PCFG

Location: Rowe 1011 Instructor: Vlado Keselj
Time: 16:05 – 17:25

Previous Lecture

- Natural language syntax:
  - phrase structure, clauses, sentences
  - Parsing, parse tree examples
- Context-Free Grammars review:
  - formal definition
  - inducing a grammar from parse trees
  - derivations, and other notions
- Bracket representation of a parse tree
- Parsing NL in Prolog using Difference Lists
- Reading: [JM] Ch 12

21.2 Definite Clause Grammar (DCG)

Basic Definite Clause Grammar (DCG)

- DCG — Prolog built-in mechanism for parsing

Example

```
s  --> np, vp.
np --> d, n.
d  --> [the].
n  --> [dog].
n  --> [dogs].
vp --> [run].
vp --> [runs].
```

This is a simplified Prolog notation for predicates described previously. The variables S, R, and I that were used for difference lists before, are now implicitly used in the DCG grammar. We can use more Prolog variables. These variables are added at the front of the predicate argument list in the implicit representation.

21.3 Building a Parse Tree in DCG

Building a Parse Tree

A parse tree can be built in the following way:

```
s(s(Tn,Tv))    --> np(Tn), vp(Tv).
np(np(Td,Tn)) --> d(Td), n(Tn).
d(d(the))      --> [the].
n(n(dog))      --> [dog].
n(n(dogs))     --> [dogs].
vp(vp(run))    --> [run].
vp(vp(runs))   --> [runs].
```

At Prolog prompt we type and obtain:

```
?- s(X, [the, dog, runs], []).
 X = s(np(d(the),n(dog)),vp(runs));
```

21.4 Example of Handling Agreement in DCG

Handling Agreement

```
s(s(Tn,Tv))       --> np(Tn,A), vp(Tv,A).
np(np(Td,Tn),A)   --> d(Td), n(Tn,A).
d(d(the))         --> [the].
n(n(dog),sg)      --> [dog].
n(n(dogs),pl)     --> [dogs].
vp(vp(run),pl)    --> [run].
vp(vp(runs),sg)   --> [runs].
```

This grammar will accept sentences “the dog runs” and “the dogs run” but not “the dog run” and “the dogs runs”. Other phenomena can be modeled in a similar fashion.

21.5 Embedded Code in DCG

Embedded Code

We can embed additional Prolog code using braces, e.g.:

```
s(T)    --> np(Tn), vp(Tv), {T = s(Tn,Tv)}.
```

and so on, is another way of building the parse tree.

22 Probabilistic Context-Free Grammar (PCFG)

Reading: Chapters 13 and 14

The main issue with the CFG model when parsing natural languages is ambiguity. A typical natural grammar that we can create to describe a natural language is ambiguous; which means that for many sentences it will give two or more parse trees and we will have a problem of choosing one of those trees, likely one that does not represent an intended meaning of the sentence. A way to solve this problem is the use of **Probabilistic Context-Free Grammar (PCFG)**, also known as the **Stochastic Context-Free Grammar (SCFG)**.

Both, n-gram model and HMM are linear models, which may not be most suitable to model the structured nature of natural language syntax. While Bayesian Networks could be one way of capturing structured nature of language

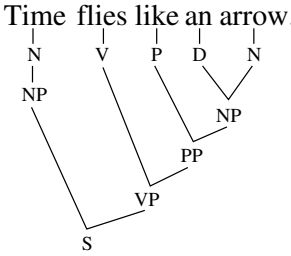
in a probabilistic way, PCFGs represent another way, which is directly derived from the Context-Free Grammar formalism.

There are arguments that the use of PCFGs could also improve language modeling. For example, if we consider the words that may continue the sentence

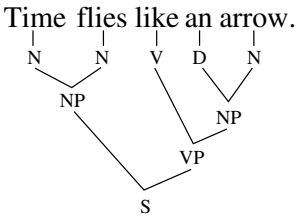
The velocity of the seismic waves rises to...

a linear model will likely assign a higher probability to the word "rise" after the plural "waves" than to the word "rises," which actually correctly appears in the sentence and agrees with the head "velocity" of the noun phrase.

As previously described, context-free grammars represent a structural model for describing syntax. For example, the syntax of the sentence "Time flies like an arrow." could be represented as the following context-free parse tree:



There are known efficient parsing algorithms for context-free grammars in the theory of of formal languages, and applications such as design of compilers and interpreters for programming languages. Two examples of such parsing approaches are recursive descent parsing and shift-reduce LR parsing. A large obstacle in applying these parsers to the problem of NL parsing is in the requirement that the language is unambiguous. Natural languages are inherently ambiguous and a parser for natural language must handle ambiguous grammars and ambiguous input. For example, if we assume a different meaning of the above sentence, we obtain a different parse tree, like the following one:



The above two trees induce the following CFG:

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			

To have a complete CFG specification, we need to add that the set of terminals is {'time', 'arrow', 'flies', 'an', 'like'}, the set of non-terminals is { S, NP, VP, D, N, PP, P, V}, and the start symbol is S.

If we parse the same sentence using this grammar, then we will obtain at least two different parse trees. To make parsing more usable, we need a way of assigning a score or probability to each tree, so we can always choose the "best" parse tree in a certain sense.

22.1 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:

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

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 $N \to \alpha$ is the probability $\mathrm{P}(N \to \alpha|N)$, so if $N \to \alpha_1$, $N \to \alpha_2$, …, $N \to \alpha_n$ are all rules with the nonterminal N on its left hand side, then

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

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

$$P(\text{tree}) = ?$$

- Generation
- Learning
- Inference
  - Marginalization

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

– Conditioning

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

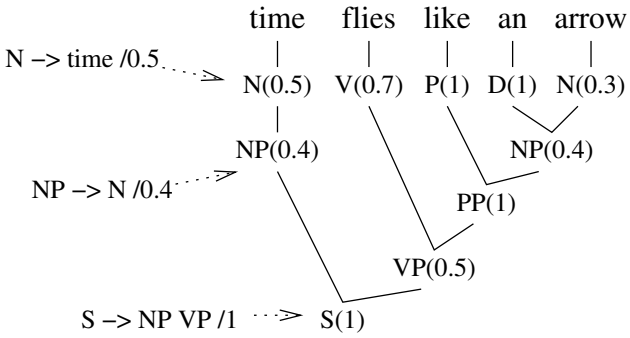
– Completion

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

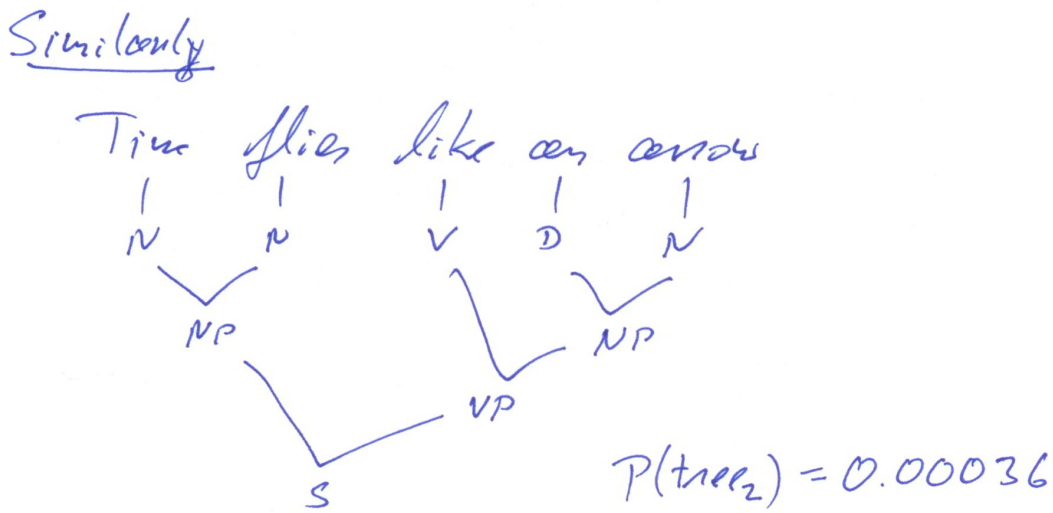
Evaluation

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.

The following tree illustrate the evaluation example using our example:



P(tree) = 0.5x0.7x1x1x0.3x0.4x0.4x1x0.5x1=0.0084



Generation

We can generate (sample) sentences by starting with the initial nonterminal S, and by rewriting it using a rule $S \to \alpha$ 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

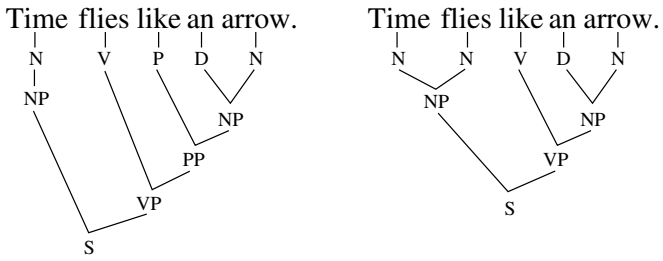
N → time / 0.5
N → arrow / 0.3
N → flies / 0.2

- 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 parse treebank is by counting the number of rule occurrences. For example, let us revisit the two parse trees of the sentence "Time flies like an arrow.", given at the beginning of this chapter:



If we extract all the rules from the non-leaf vertices of these trees, and sort them by the left hand side, we obtain the

following list:

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

Now, instead of repeating the same rules, we can only include them once and record their counts like this:

D	→	an	$\times 2$	N	→	arrow	$\times 2$	S	→	NP VP	$\times 2$
NP	→	D N	$\times 2$	N	→	flies	$\times 1$	VP	→	V NP	$\times 1$
NP	→	N N	$\times 1$	N	→	time	$\times 2$	VP	→	V PP	$\times 1$
NP	→	N	$\times 1$	PP	→	P NP	$\times 1$	V	→	flies	$\times 1$
				P	→	like	$\times 1$	V	→	like	$\times 1$

We can not calculate total counts for all rules with the same left-hand side and obtain totals: 2 for D, 4 for NP, 5 for N, 1 for PP, 1 for P, 2 for S, 2 for VP, and 2 for V. Finally, we calculate probabilities of all rules by dividing their counts with corresponding total counts for the corresponding left-hand side non-terminal:

D	→	an	$/1$	N	→	arrow	$/0.4$	S	→	NP VP	$/1$
NP	→	D N	$/0.5$	N	→	flies	$/0.2$	VP	→	V NP	$/0.5$
NP	→	N N	$/0.25$	N	→	time	$/0.4$	VP	→	V PP	$/0.5$
NP	→	N	$/0.25$	PP	→	P NP	$/1$	V	→	flies	$/0.5$
				P	→	like	$/1$	V	→	like	$/0.5$

This final list of rules with their probabilities is the PCFG *learned* from the two parse trees given earlier. A set of parse trees is called a *treebank*, particularly if it is large and manually curated, so the two given parse trees can be called a small treebank. We can also say that this final PCFG is *induced* or *generated* by the given treebank.

Inference

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

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

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

22.2 Expressing PCFGs in DCGs

Expressing PCFGs in DCGs

Let us consider the previous example of a PCFG:

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	$/1$								

The probabilities can be calculated as an addition argument:

```
s(T,P) --> np(T1,P1), vp(T2,P2),
           {T = s(T1,T2), P is P1 * P2 * 1}.
np(T,P) --> n(T1,P1), {T = n(T1), P is P1 * 0.4}.
```

and so on.

A full DCG code for the above PCFG could be:

```
s(s(Tn,Tv),P) --> np(Tn,P1), vp(Tv,P2), {P is P1 * P2}.
np(np(T),P) --> n(T,P1), {P is P1 * 0.4}.
np(np(T1,T2),P) --> n(T1,P1), n(T2,P2), {P is P1 * P2 * 0.2}.
np(np(Td,Tn),P) --> d(Td,P1), n(Tn,P2), {P is P1 * P2 * 0.4}.
v(v(like), 0.3) --> [like].
v(v(flies), 0.7) --> [flies].
p(p(like), 1.0) --> [like].
vp(vp(Tv,Tn), P) --> v(Tv, P1), np(Tn, P2), {P is P1 * P2 * 0.5}.
vp(vp(Tv,Tp), P) --> v(Tv, P1), pp(Tp, P2), {P is P1 * P2 * 0.5}.
pp(pp(Tp,Tn), P) --> p(Tp, P1), np(Tn, P2), {P is P1 * P2}.
n(n(time), 0.5) --> [time].
n(n(arrow), 0.3) --> [arrow].
n(n(flies), 0.2) --> [flies].
d(d(an), 1.0) --> [an].
```

After loading this grammar into Prolog interpreter, and then after issuing the query:
?- `s(T,P,[time,flies,like,an,arrow],[]).`
the interpreter would reply with:

```
T = s(np(n(time)), vp(v(flies), pp(p(like), np(d(an), n(arrow)))))
P = 0.0084
```

and after typing `;` (semi-colon), we get:

```
T = s(np(n(time), n(flies)), vp(v(like), np(d(an), n(arrow))))
P = 0.00036
```

After typing second '`;`', the interpreter reports 'No' since there are no more parse trees.