The CKY Parsing Algorithm and PCFGs

COMP-599

Oct 13, 2015

Outline

CYK parsing

PCFGs

Probabilistic CYK parsing

CFGs and Constituent Trees

Rules/productions:

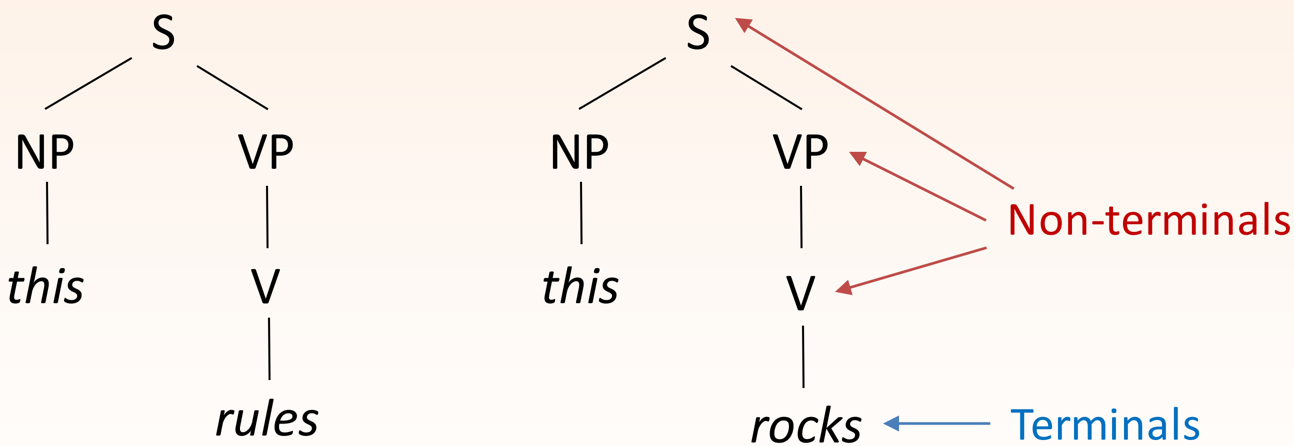
S → NP VP

NP → *this*

VP → V

V → *is* | *rules*| *jumps* | *rocks*

Trees:



Parsing into a CFG

Given:

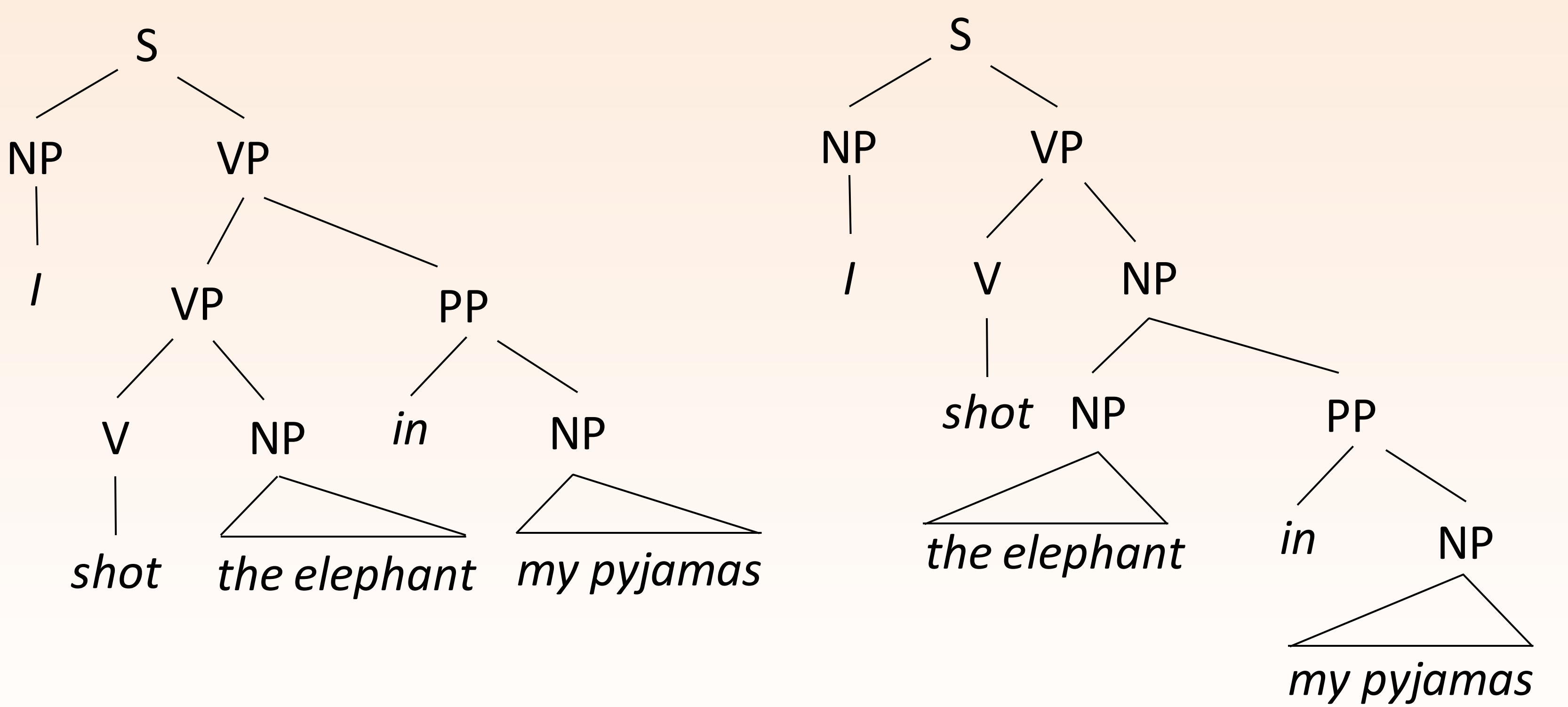
1. CFG
2. A sentence made up of words that are in the terminal vocabulary of the CFG

Task: Recover all possible parses of the sentence.

Why *all* possible parses?

Syntactic Ambiguity

I shot the elephant in my pyjamas.



CYK Algorithm

Cocke-Younger-Kasami algorithm

- A dynamic programming algorithm – partial solutions are stored and efficiently reused to find all possible parses for the entire sentence.
- Also known as the CKY algorithm

Steps:

1. Convert CFG to an appropriate form
2. Set up a table of possible constituents
3. Fill in table
4. Read table to recover all possible parses

Chomsky Normal Form

To make things easier later, need all productions to be in one of these forms:

1. A $\rightarrow$ B C, where A, B, C are nonterminals
2. A $\rightarrow$ *s*, where A is a non-terminal *s* is a terminal

This is actually not a big problem.

Converting to CNF (1)

Rule of type A → B C D …

- Rewrite into: A → X1 D … and X1 → B C

Rule of type A → *s* B

- Rewrite into: A → X2 B and X2 → *s*

Rule of type A → B

- Everywhere in which we see B on the LHS, replace it with A

Examples of Conversion

Let's convert the following grammar fragment into CNF:

S → NP VP

VP → V NP PP

VP → V NP

NP → N

NP → Det N

NP → Det N PP

PP → *in* NP

N → *I* | *elephant* | *pyjamas*

V → *shot*

Det → *my* | *the*

Next: Set Up a Table

This table will store all of the constituents that can be built from contiguous spans within the sentence.

Let sentence have N words. w[0], w[1], … w[N-1]

Create table, such that a cell in row i column j corresponds to the span from w[i:j+1], zero-indexed.

- Since i < j, we really just need half the table.

The entry at each cell is a list of non-terminals that can span those words according to the grammar.

Parse Table

I$_0$	*shot*$_1$	*the*$_2$	*elephant*$_3$	*in*$_4$	*my*$_5$	*pyjamas*$_6$
[0:1]	[0:2]	[0:3]	[0:4]	[0:5]	[0:6]	[0:7]
	[1:2]	[1:3]	[1:4]	[1:5]	[1:6]	[1:7]
		[2:3]	[2:4]	[2:5]	[2:6]	[2:7]
			[3:4]	[3:5]	[3:6]	[3:7]
				[4:5]	[4:6]	[4:7]
					[5:6]	[5:7]
						[6:7]

S → NP VP
VP → X1 PP
VP → V NP
NP → Det N
NP → X2 PP
PP → P NP
P → *in*
NP → *I* | *elephant* | *pyjamas*
N → *I* | *elephant* | *pyjamas*
V → *shot*
Det → *my* | *the*

X1 → V NP
X2 → Det N

Filling in Table: Base Case

One word (e.g., cell [0:1])

- Easy – add all the lexical rules that can generate that word

Base Case Examples (First 3 Words)

I_0	shot_1	the_2	$\mathit{elephant}_3$	in_4	my_5	$\mathit{pyjamas}_6$
[0:1] NP N	[0:2]	[0:3]	[0:4]	[0:5]	[0:6]	[0:7]
	[1:2] V	[1:3]	[1:4]	[1:5]	[1:6]	[1:7]
		[2:3] Det	[2:4]	[2:5]	[2:6]	[2:7]
			[3:4]	[3:5]	[3:6]	[3:7]
				[4:5]	[4:6]	[4:7]
					[5:6]	[5:7]
						[6:7]

S → NP VP
VP → X1 PP
X1 → V NP
VP → V NP
NP → Det N
NP → X2 PP
X2 → Det N
PP → P NP
P → *in*
NP → *I* | *elephant* | *pyjamas*
N → *I* | *elephant* | *pyjamas*
V → *shot*
Det → *my* | *the*

Filling in Table: Recursive Step

Cell corresponding to multiple words

- eg., cell for span [0:3] *I shot the*
- Key idea: all rules that produce phrases are of the form
  A → B C
- So, check all the possible break points *m* in between the start *i* and the end *j*, and see if we can build a constituent with a rule in the form, A [*i*:*j*] → B [*i*:*m*] C [*m*:*j*]

Recurrent Step Example 1

I$_0$	*shot*$_1$	*the*$_2$	*elephant*$_3$	*in*$_4$	*my*$_5$	*pyjamas*$_6$
[0:1] NP N	[0:2] ?	[0:3]	[0:4]	[0:5]	[0:6]	[0:7]
	[1:2] V	[1:3]	[1:4]	[1:5]	[1:6]	[1:7]
		[2:3]	[2:4]	[2:5]	[2:6]	[2:7]
			[3:4]	[3:5]	[3:6]	[3:7]
				[4:5]	[4:6]	[4:7]
					[5:6]	[5:7]
						[6:7]

S $\rightarrow$ NP VP
VP $\rightarrow$ X1 PP
VP $\rightarrow$ V NP
NP $\rightarrow$ Det N
NP $\rightarrow$ X2 PP
PP $\rightarrow$ P NP
P $\rightarrow$ *in*
NP $\rightarrow$ *I* | *elephant* | *pyjamas*
N $\rightarrow$ *I* | *elephant* | *pyjamas*
V $\rightarrow$ *shot*
Det $\rightarrow$ *my* | *the*

X1 $\rightarrow$ V NP
X2 $\rightarrow$ Det N

Recurrent Step Example 2

I$_0$	*shot*$_1$	*the*$_2$	*elephant*$_3$	*in*$_4$	*my*$_5$	*pyjamas*$_6$
[0:1] NP N	[0:2]	[0:3]	[0:4]	[0:5]	[0:6]	[0:7]
	[1:2] V	[1:3]	[1:4]	[1:5]	[1:6]	[1:7]
		[2:3] Det	[2:4] ?	[2:5]	[2:6]	[2:7]
			[3:4] NP N	[3:5]	[3:6]	[3:7]
				[4:5]	[4:6]	[4:7]
					[5:6]	[5:7]
						[6:7]

S → NP VP
VP → X1 PP
VP → V NP
NP → Det N
NP → X2 PP
PP → P NP
P → *in*
NP → *I* | *elephant* | *pyjamas*
N → *I* | *elephant* | *pyjamas*
V → *shot*
Det → *my* | *the*

X1 → V NP
X2 → Det N

Backpointers

I_0	$shot_1$	the_2	$elephant_3$	in_4	my_5	$pyjamas_6$
[0:1] NP N	[0:2]	[0:3]	[0:4]	[0:5]	[0:6]	[0:7]
	[1:2] V	[1:3]	[1:4]	[1:5]	[1:6]	[1:7]
		[2:3] Det	[2:4] NP	[2:5]	[2:6]	[2:7]
			[3:4] NP N	[3:5]	[3:6]	[3:7]
				[4:5]	[4:6]	[4:7]
					[5:6]	[5:7]
						[6:7]

S → NP VP
VP → X1 PP   X1 → V NP
VP → V NP
NP → Det N
NP → X2 PP   X2 → Det N
PP → P NP
P → *in*
NP → *I* | *elephant* | *pyjamas*
N → *I* | *elephant* | *pyjamas*
V → *shot*
Det → *my* | *the*

Store where you came from!

Putting It Together

I$_0$	*shot*$_1$	*the*$_2$	*elephant*$_3$	*in*$_4$	*my*$_5$	*pyjamas*$_6$
NP N [0:1]	[0:2]	[0:3]	[0:4]	[0:5]	[0:6]	[0:7]
	V [1:2]	[1:3]	[1:4]	[1:5]	[1:6]	[1:7]
		Det [2:3]	NP (Det 2:3 ,N 3:4) [2:4]	[2:5]	[2:6]	[2:7]
			NP N [3:4]	[3:5]	[3:6]	[3:7]
				[4:5]	[4:6]	[4:7]
					[5:6]	[5:7]
						[6:7]

S → NP VP
VP → X1 PP   X1 → V NP
VP → V NP
NP → Det N
NP → X2 PP   X2 → Det N
PP → P NP
P → *in*
NP → *I* | *elephant* | *pyjamas*
N → *I* | *elephant* | *pyjamas*
V → *shot*
Det → *my* | *the*

Fill the table in the correct order!

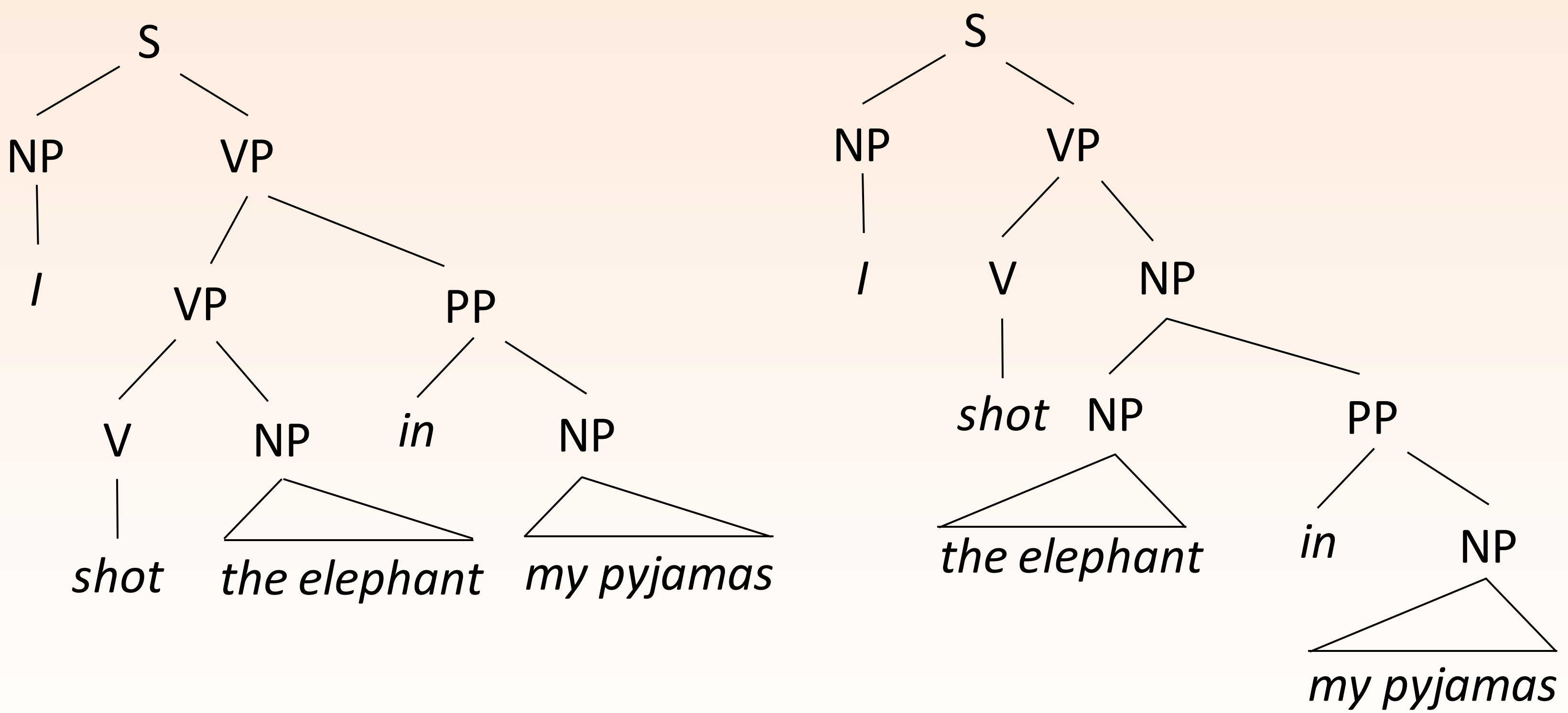
Finish the Example

Let’s finish the example together for practice

How do we reconstruct the parse trees from the table?

Dealing with Syntactic Ambiguity

In practice, one of these is more like than the other:



How to distinguish them?

Probabilistic CFGs

Associate each rule with a probability:

e.g.,	
NP → NP PP	0.2
NP → Det N	0.4
NP → *I*	0.1
…	
V → *shot*	0.005

Probability of a parse tree for a sentence is the product of the probabilities of the rules in the tree.

Formally Speaking

For each nonterminal $A \in N$,

$$\sum_{\alpha \to \beta \in R \text{ s.t.} \alpha = A} \Pr(\alpha \to \beta) = 1$$

- i.e., rules for each LHS form a probability distribution

If a tree t contains rules $\alpha_1 \to \beta_1, \alpha_2 \to \beta_2, \ldots,$

$$\Pr(t) = \prod_i \Pr(\alpha_i \to \beta_i)$$

- Tree probability is product of rule probabilities

Probabilistic Parsing

Goal: recover the best parse for a sentence, along with its probability

For a sentence, sent,

let $\tau(\text{sent})$ be the set of possible parses for it,

we want to find

$$\underset{t \in \tau(\text{sent})}{\operatorname{argmax}} \Pr(t)$$

Idea: extend CYK to keep track of probabilities in table

Extending CYK to PCFGs

Previously, cell entries are nonterminals (+ backpointer)

e.g., table[2:4] = {{NP, Det[2:3] N[3:4] }}

table[3:4] = {{NP, } {N, }}

Now, cell entries include the (best) probability of generating the constituent with that non-terminal

e.g., table[2:4] = {{NP, Det[2:3] N[3:4], 0.215}}

table[3:4] = {{NP, , 0.022} {N, , 0.04}}

Equivalently, write as 3-dimensional array

table[2, 4, NP] = 0.215 (Det[2:3], N[3:4])

table[3, 4, NP] = 0.022

table[3, 4, N] = 0.04

New Recursive Step

Filling in dynamic programming table proceeds almost as before.

During recursive step, compute probability of new constituents to be constructed:

Pr(A[i:j]→B[i:m] C[m:j]) = Pr(A→BC) × table[i,m,B] × table[m,j,C]

From PCFG

From previously filled cells

There could be multiple rules that form constituent A for span [i:j]. Take max:

table[i,j,A] =

$$\max_{A \to BC,\ \text{break at}\ m} \Pr(A[i:j] \to B[i:m]C[m:j])$$

Example

I_0	$shot_1$	the_2	$elephant_3$	in_4	my_5	$pyjamas_6$
[0:1] NP, 0.25 N, 0.625	[0:2]	[0:3]	[0:4]	[0:5]	[0:6]	[0:7]
	[1:2] V, 1.0	[1:3]	[1:4]	[1:5]	[1:6]	[1:7]
		[2:3] Det, 0.6	[2:4] NP, ?	[2:5]	[2:6]	[2:7]
			[3:4] NP, 0.1 N, 0.25	[3:5]	[3:6]	[3:7]
				[4:5]	[4:6]	[4:7]
					[5:6]	[5:7]
						[6:7]

New value:

$0.6 * 0.25 * \Pr(NP \rightarrow Det\ N)$

Bottom-Up vs. Top-Down

CYK algorithm is **bottom-up**

- Starting from words, build little pieces, then big pieces

Alternative: **top-down** parsing

- Starting from the start symbol, expand non-terminal symbols according to rules in the grammar.
- Doing this efficiently can also get us all the parses of a sentence (**Earley algorithm**)

How to Train a PCFG?

Derive from a treebank, such as WSJ.

Simplest version:

- each LHS corresponds to a categorical distribution
- outcomes of the distributions are the RHS
- MLE estimates:

$$\Pr(\alpha \to \beta) = \frac{\#(\alpha \to \beta)}{\#\alpha}$$

- Can smooth these estimates in various ways, some of which we've discussed