Foundations of Natural Lanugage Processing
Lecture 12
CKY Parsing, treebanks and statistical parsing

Shay Cohen
(slides from Henry Thompson, Alex Lascarides and Sharon Goldwater)

28 March 2020

School of informatics

CKY Algorithm

CKY (Cocke, Kasami, Younger) is a **bottom-up**, **breadth-first** parsing algorithm.

- Original version assumes grammar in Chomsky Normal Form.
- Add constituent A in cell (i, j) if there is:
  - a rule $A \rightarrow B$, and a B in cell (i, j), **or**
  - a rule $A \rightarrow B\ C$, and a B in cell (i,k) and a C in cell (k, j).

CKY Algorithm

CKY (Cocke, Kasami, Younger) is a **bottom-up**, **breadth-first** parsing algorithm.

- Original version assumes grammar in Chomsky Normal Form.
- Add constituent A in cell (i, j) if there is:
  - a rule $A \rightarrow B$, and a B in cell (i, j), **or**
  - a rule $A \rightarrow B\ C$, and a B in cell (i,k) and a C in cell (k, j).
- Fills chart in order: only looks for rules that use a constituent of length n **after** finding all constituents of length less than n. So, guaranteed to find all possible parses.

CKY Pseudocode

- Assume input sentence with indices 0 to n, and chart c.

```
for len = 1 to n:   #number of words in constituent
   for i = 0 to n-len:   #start position
      j = i+len   #end position
      #process unary rules
      foreach A->B where c[i,j] has B
         add A to c[i,j] with a pointer to B
      for k = i+1 to j-1   #mid position
         #process binary rules
         foreach A->B C where c[i,k] has B and c[k,j] has C
            add A to c[i,j] with pointers to B and C
```

- Takes time $O(Gn^3)$, where G is the number of grammar rules.

CKY Example

S → NP VP
NP → D N | Pro | PropN
D → PosPro | Art | NP 's
VP → Vi | Vt NP | Vp NP VP
Pro → i | we | you | he | she | him | her
PosPro → my | our | your | his | her
PropN → Robin | Jo
Art → a | an | the
N → cat | dog | duck | saw | park | telescope | bench
Vi → sleep | run | duck
Vt → eat | break | see | saw
Vp → see | saw | heard

CKY Example

Length 1 constituents: POS tags

	1	2	3	4
0	Pro			
1		Vt,Vp,N		
2			Pro, PosPro	
3				N,Vi
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

- We've added all POSs that are allowed for each word.

CKY Example

Length 1 constituents: Unary rule NP → Pro

	1	2	3	4
0	Pro, NP			
1		Vt,Vp,N		
2			Pro, PosPro	
3				N,Vi
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

- red shows which children create which parents.
- Normally we'd add pointers from parent to child to store this info permanently, but we'd end up with too many arrows here to see what's going on!

CKY Example

Length 1 constituents: Unary rules D → PosPro, NP → Pro and VP → Vi

	1	2	3	4
0	Pro, NP			
1		Vt,Vp,N		
2			Pro, NP, PosPro, D	
3				N, Vi, VP
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

- red shows which children create which parents.
- Normally we'd add pointers from parent to child to store this info permanently, but we'd end up with too many arrows here to see what's going on!

CKY Example

Length 2 constituents: Binary rule NP → D N

	1	2	3	4
0	Pro, NP			
1		Vt,Vp,N		
2			Pro, NP, PosPro, D	NP
3				N,Vi, VP
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

- red shows which children create which parents.
- Normally we'd add pointers from parent to child to store this info permanently, but we'd end up with too many arrows here to see what's going on!

CKY Example

Length 3 constituents: Binary rule VP → Vt NP

	1	2	3	4
0	Pro, NP			
1		Vt,Vp,N		VP
2			Pro, N, PosPro, D	NP
3				N,Vi, VP
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

- Vt from (1,2) plus NP from (2,4) makes a VP from (1,4).
- For cell (1,4) we also consider (1,3) plus (3,4) but there's nothing in those cells that can combine to make a larger phrase.

CKY Example

Length 3 constituents: alternate parses

	1	2	3	4
0	Pro, NP			
1		Vt,Vp,N		VP
2			Pro, NP, PosPro, D	NP
3				N, Vi, VP
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

- We also have another way to build the same VP (1,4). Add more pointers to remember this new analysis.
- (Not standard CKY because we used a ternary rule! In reality we would have converted this rule into CNF, but still ended up with two parses for VP.)

CKY Example

Length 4 constituents: Binary rule S → NP VP

	1	2	3	4
0	Pro, NP			S
1		Vt,Vp,N		VP
2			Pro, PosPro, D	NP
3				N,Vi
	$_0$he$_1$	$_1$saw$_2$	$_2$her$_3$	$_3$duck$_4$

- When we build the S, it doesn't matter anymore that there are two VP analyses, we just see the VP.
  - Cells contain *sets* of labels
  - Or dicts, if we are keeping backpointers
- Ambiguity is only clear if we go on to reconstruct the parses using our backpointers.

A note about CKY ordering

- Notice that to fill cell (i, j), we use a cell from row i and a cell from column j.
- So, before trying to fill (i, j) we must fill in all cells to the left of j in row i and all cells below cell i in column j.
- Here, we filled in all short entries, then longer ones
  - effectively sweeping out diagonals beginning with the main diagonal and moving up to the right
- but other orders can work (e.g., J&M fill in all spans ending at j, then increment j.)

CKY in practice

- Avoids re-computing substructures, so much more efficient than depth-first parsers (in worst case).
- Still may compute a lot of unnecessary partial parses.
- Simple version requires converting the grammar to CNF (may cause blowup: remember time depends on grammar too!).

Various other chart parsing methods avoid these issues by combining top-down and bottom-up approaches (see J&M or recall Inf2A).

But rather than going that way, we'll focus on **statistical parsing** which can help deal with both ambiguity and efficiency issues.

Towards probabilistic parsing

- We've seen various parsing algorithms, most recently CKY, that parses exhaustively in polynomial time.
- But we haven't discussed how to choose which of many possible parses is the right one.
- The obvious solution: probabilities.

How big a problem is disambiguation?

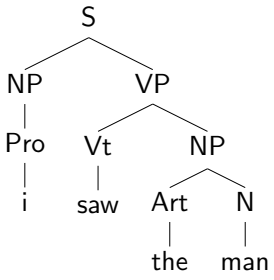
- Early work in computational linguistics tried to develop broad-coverage hand-written grammars.
  - That is, grammars that *include* all sentences humans would judge as grammatical in their language;
  - while *excluding* all other sentences.
- As coverage grows, sentences can have hundreds or thousands of parses. Very difficult to write heuristic rules for disambiguation.
- Plus, grammar is hard to keep track of! Trying to fix one problem can introduce others.
- Enter the **treebank grammar**.

Treebank grammars

- The big idea: instead of paying linguists to write a grammar, pay them to annotate real sentences with parse trees.
- This way, we implicitly get a grammar (for CFG: read the rules off the trees).
- **And** we get probabilities for those rules (using any of our favorite estimation techniques).
- We can use these probabilities to improve disambiguation and even speed up parsing.

Treebank grammars

For example, if we have this tree in our corpus:



Then we add rules

$\texttt{S} \rightarrow \texttt{NP VP}$
$\texttt{NP} \rightarrow \texttt{Pro}$
$\texttt{Pro} \rightarrow \texttt{i}$
$\texttt{VP} \rightarrow \texttt{Vt NP}$
$\texttt{Vt} \rightarrow \texttt{saw}$
$\texttt{NP} \rightarrow \texttt{Art N}$
$\texttt{Art} \rightarrow \texttt{the}$
$\texttt{N} \rightarrow \texttt{man}$

With more trees, we can start to count rules and estimate their probabilities.

Example: The Penn Treebank

- The first large-scale parse annotation project, begun in 1989.
- Original corpus of syntactic parses: Wall Street Journal text
  - About 40,000 annotated sentences (1m words)
  - Standard phrasal categories like `S`, `NP`, `VP`, `PP`.
- Now many other data sets (e.g. transcribed speech), and different kinds of annotation; also inspired treebanks in many other languages.

Creating a treebank PCFG

A **probabilistic context-free grammar** (PCFG) is a CFG where each rule $\texttt{NT} \rightarrow \beta$ (where β is a symbol sequence) is assigned a probability $P(\beta|NT)$.

- The sum over all expansions of `NT` must equal 1: $\sum_{\beta'} P(\beta'|NT) = 1$.
- Easiest way to create a PCFG from a treebank: MLE
  - Count all occurrences of $\texttt{NT} \rightarrow \beta$ in treebank.
  - Divide by the count of all rules whose LHS is `NT` to get $P(\beta|NT)$
  - $P(NT \longrightarrow C_1, C_2 \ldots C_n | NT) = \frac{count(NT \longrightarrow C_1, C_2 \ldots C_n)}{count(NT)}$
- But as usual many rules have very low frequencies, so MLE isn't good enough and we need to smooth.

The probability of a parse

- Under this model, the probability of a parse t is simply the product of all rules in the parse:

$$P(t) = \prod_{NT \to \beta \in t} NT \to \beta$$

Statistical disambiguation example

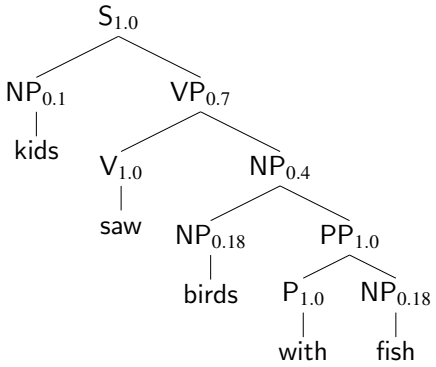
How can parse probabilities help disambiguate PP attachment?

- Let's use the following PCFG, inspired by Manning & Schuetze (1999):

S → NP VP	1.0	NP → NP PP	0.4
PP → P NP	1.0	NP → kids	0.1
VP → V NP	0.7	NP → birds	0.18
VP → VP PP	0.3	NP → saw	0.04
P → with	1.0	NP → fish	0.18
V → saw	1.0	NP → binoculars	0.1

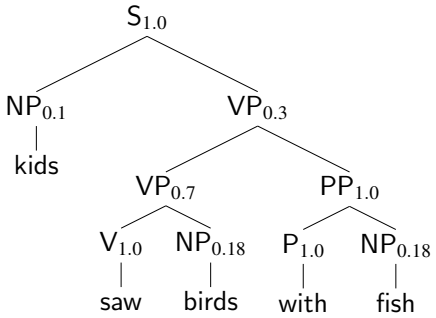
- We want to parse kids saw birds with fish.

Probability of parse 1



- $P(t_1) = 1.0 \cdot 0.1 \cdot 0.7 \cdot 1.0 \cdot 0.4 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0009072$

Probability of parse 2



- $P(t_2) = 1.0 \cdot 0.1 \cdot 0.3 \cdot 0.7 \cdot 1.0 \cdot 0.18 \cdot 1.0 \cdot 1.0 \cdot 0.18 = 0.0006804$
- which is less than $P(t_1) = 0.0009072$, so t_1 is preferred. Yay!

The probability of a sentence

- Since t implicitly includes the words $\vec{w}$, we have $P(t) = P(t, \vec{w})$.
- So, we also have a **language model**. Sentence probability is obtained by summing over $T(\vec{w})$, the set of valid parses of $\vec{w}$:

$$P(\vec{w}) = \sum_{t \in T(\vec{w})} P(t, \vec{w}) = \sum_{t \in T(\vec{w})} P(t)$$

- In our example,
  $P(\text{kids saw birds with fish}) = 0.0006804 + 0.0009072$.

Probabilistic CKY

It is straightforward to extend CKY parsing to the probabilistic case.

- Goal: return the highest probability parse of the sentence (analogous to Viterbi)
  - When we find an `NT` spanning (i, j), store its probability along with its label in cell (i, j).
  - If we later find an `NT` with the same span but higher probability, replace the probability *and the backpointers* for `NT` in cell (i, j).

Probabilistic CKY

We also have analogues to the other HMM algorithms.

- The **inside algorithm** computes the probability of the sentence (analogous to forward algorithm)
  - Same as above, but instead of storing the *best* parse for `NT`, store the *sum* of all parses.
- The **inside-outside algorithm** algorithm is a form of EM that learns grammar rule probs from unannotated sentences (analogous to forward-backward).

Best-first probabilistic parsing

- So far, we've been assuming **exhaustive** parsing: return all possible parses.
- But treebank grammars are huge!!
  - Exhaustive parsing of WSJ sentences up to 40 words long adds on average over 1m items to chart per sentence.[1]
  - Can be hundreds of possible parses, but most have extremely low probability: do we really care about finding these?
- **Best-first** parsing can help.

[1] Charniak, Goldwater, and Johnson, WVLC 1998.

Best-first probabilistic parsing

Basic idea: use probabilities of subtrees to decide which ones to build up further.

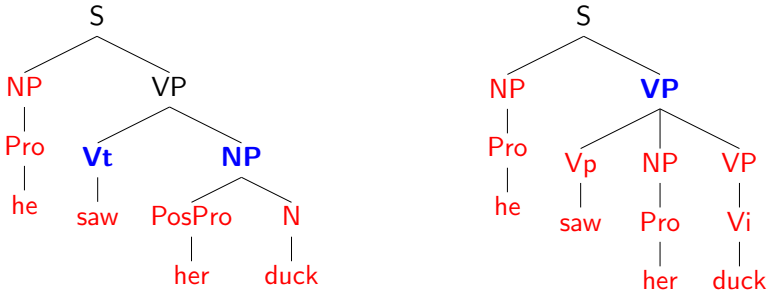
- Each time we find a new constituent, we give it a **score** ("figure of merit") and add it to an **agenda**, which is ordered by score.
- Then we pop the next item off the agenda, add it to the chart, and see which new constituents we can make using it.
- We add those to the agenda, and iterate.

Notice we are no longer filling the chart in any fixed order.

Many variations on this idea (including incremental ones), often limiting the size of the agenda by **pruning** out low-scoring edges (**beam search**).

Best-first intuition

Suppose red constituents are in chart already; **blue** are on agenda.



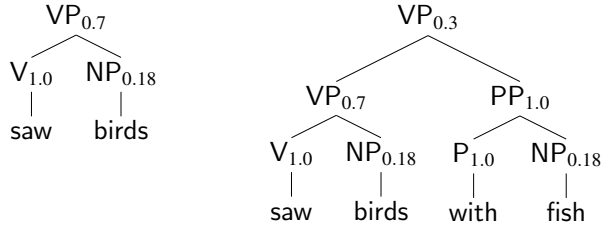
If the VP in right-hand tree scores high enough, we'll pop that next, add it to chart, then find the S. So, we could complete the whole parse before even finding the alternative VP.

How do we score constituents?

Perhaps according to the probability of the subtree they span? So, P(left example)=(0.7)(0.18) and P(right example)=0.18.

How do we score constituents?

But what about comparing different sized constituents?



A better figure of merit

- If we use raw probabilities for the score, **smaller** constituents will almost always have higher scores.
  - Meaning we pop all the small constituents off the agenda before the larger ones.
  - Which would be very much like exhaustive bottom-up parsing!
- Instead, we can divide by the **number of words** in the constituent.
  - Very much like we did when comparing language models (recall **per-word** cross-entropy)!
- This works much better, but now not guaranteed to find the best parse first. Other improvements are possible (including A*).

But wait a minute...

Best-first parsing shows how simple ("vanilla") treebank PCFGs can improve **efficiency**. But do they really solve the problem of disambiguation?

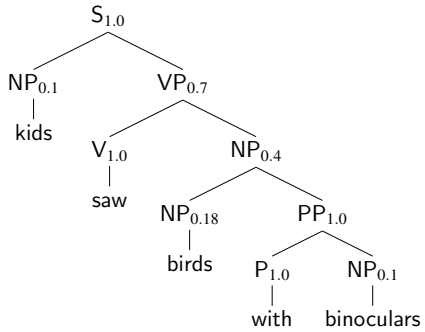
- Our example grammar gave the right parse for this sentence:

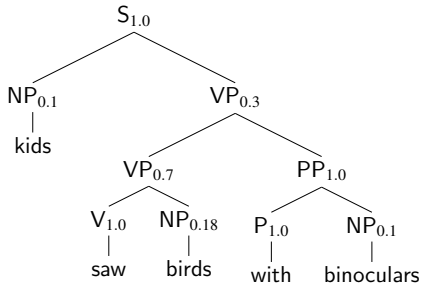
  kids saw birds with fish

- What happens if we parse this sentence?

  kids saw birds with binoculars

Vanilla PCFGs: no lexical dependencies





- Exactly the same probabilities as the "fish" trees, except divide out $P(\text{fish}|\texttt{NP})$ and multiply in $P(\text{binoculars}|\texttt{NP})$ in each case.
- So, the same (left) tree is preferred, but now incorrectly!

Vanilla PCFGs: no lexical dependencies

Replacing one word with another with the same POS will never result in a different parsing decision, even though it should!

- More examples:
  - She stood by the door covered in tears vs.
    She stood by the door covered in ivy
  - She called on the student vs.
    She called on the phone.
    (assuming "on" has the same POS...)

Vanilla PCFGs: no global structural preferences

- Ex. in Switchboard corpus, the probability of NP → Pronoun
  - in **subject position** is 0.91 — he saw the dog
  - in **object position** is 0.34 — the dog bit him

- Lots of other rules also have different probabilities depending on where they occur in the sentence.

- But PCFGs are context-free, so an NP is an NP is an NP, and will have the same expansion probs regardless of where it appears.

- Next week: How to bring words back in to the story...