Natural Language Processing

Lecture 17.2—3/12/2015

Martha Palmer

Back to CFG Parsing

- We left CFG parsing (CKY) with the problem of selecting the "right" parse out of all the possible parses...
- Now if we reduce "right" parse to "most probable parse" we get our old friend
  - Argmax P(Parse|Words)

Probabilistic CFGs

1 The framework (Model)
- How to assign probabilities to parse trees

2 Training the model (Learning)
- How to acquire estimates for the probabilities specified by the model

3 Parsing with probabilities (Decoding)
- Given an input sentence and a model how can we efficiently find the best (or *N* best) tree(s) for that input

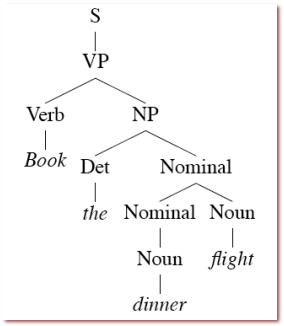
Simple Probability Model

- A derivation (tree) consists of the collection of grammar rules that are in the tree
  - The probability of a tree is the product of the probabilities of the rules in the derivation.

$$P(T,S) = \prod_{node \in T} P(rule(n))$$

Example

- How many "rules" are in this derivation?



Rule Probabilities

- So... What's the probability of a rule?
- Start at the top...
  - A tree should have an *S* at the top. So given that we know we need an *S*, we can ask about the probability of each particular *S* rule in the grammar.
    - That is P(particular S rule | S is what I need)
- So in general we need

$$P(\alpha \rightarrow \beta \mid \alpha)$$

For each rule in the grammar

Training the Model

- We can get the estimates we need from an annotated database (i.e., a treebank)

$$P(\alpha \rightarrow \beta|\alpha) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\sum_{\gamma}\text{Count}(\alpha \rightarrow \gamma)} = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)}$$

  - For example, to get the probability for a particular *VP* rule, just count all the times the rule is used and divide by the number of *VP*s overall.

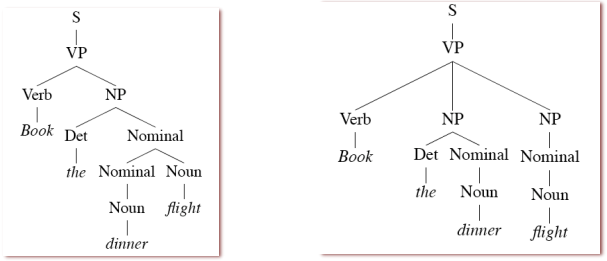
Parsing (Decoding)

- So to get the best (most probable) parse for a given input

1. Enumerate all the trees for a sentence
2. Assign a probability to each using the model
3. Return the best (argmax)

Example

- Consider...
  - *Book the dinner flight*



Examples

- These trees consist of the following rules.

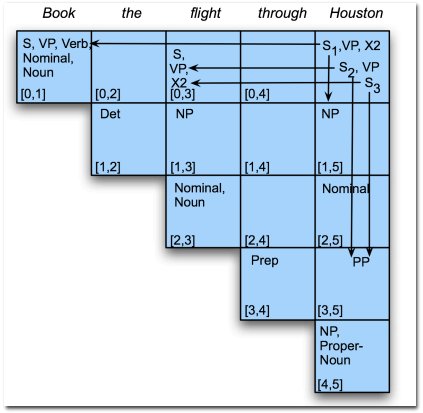
			P				P
S	→	VP	.05	S	→	VP	.05
VP	→	Verb NP	.20	VP	→	Verb NP NP	.10
NP	→	Det Nominal	.20	NP	→	Det Nominal	.20
Nominal	→	Nominal Noun	.20	NP	→	Nominal	.15
Nominal	→	Noun	.75	Nominal	→	Noun	.75
				Nominal	→	Noun	.75
Verb	→	book	.30	Verb	→	book	.30
Det	→	the	.60	Det	→	the	.60
Noun	→	dinner	.10	Noun	→	dinner	.10
Noun	→	flights	.40	Noun	→	flights	.40

$$P(T_{left}) = .05 * .20 * .20 * .20 * .75 * .30 * .60 * .10 * .40 = \mathbf{2.2 \times 10^{-6}}$$
$$P(T_{right}) = .05 * .10 * .20 * .15 * .75 * .75 * .30 * .60 * .10 * .40 = \mathbf{6.1 \times 10^{-7}}$$

Dynamic Programming

- Of course, as with normal parsing we don't really want to do it that way...
- Instead, we need to exploit dynamic programming
  - For the parsing (as with CKY)
  - And for computing the probabilities and returning the best parse (as with Viterbi and HMMs)

Example



Probabilistic CKY

- Alter CKY so that the probabilities of constituents are stored in the table as they are derived
  - Probability of a new constituent *A* derived from the rule $A \rightarrow B\ C$:
    - $P(A \rightarrow B\ C \mid A) * P(B) * P(C)$
    - Where P(B) and P(C) are already in the table given the way that CKY operates
    - What we store is the MAX probability over all the A rules.

Probabilistic CKY

function PROBABILISTIC-CKY(*words*,*grammar*) **returns** most probable parse and its probability

for $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
  for all $\{ A \mid A \rightarrow words[j] \in grammar \}$
    $table[j-1, j, A] \leftarrow P(A \rightarrow words[j])$
  for $i \leftarrow$ **from** $j-2$ **downto** 0 **do**
    for $k \leftarrow i+1$ **to** $j-1$ **do**
      for all $\{ A \mid A \rightarrow BC \in grammar,$
        and $table[i,k,B] > 0$ **and** $table[k,j,C] > 0 \}$
        if $(table[i,j,A] < P(A \rightarrow BC) \times table[i,k,B] \times table[k,j,C])$ **then**
          $table[i,j,A] \leftarrow P(A \rightarrow BC) \times table[i,k,B] \times table[k,j,C]$
          $back[i,j,A] \leftarrow \{k, B, C\}$
return BUILD_TREE($back[1,$ LENGTH(*words*), $S]), table[1,$ LENGTH(*words*), $S]$

For a detailed example

- Chris Manning on YouTube:
- https://www.youtube.com/watch?v=hq80J8kBg-Y

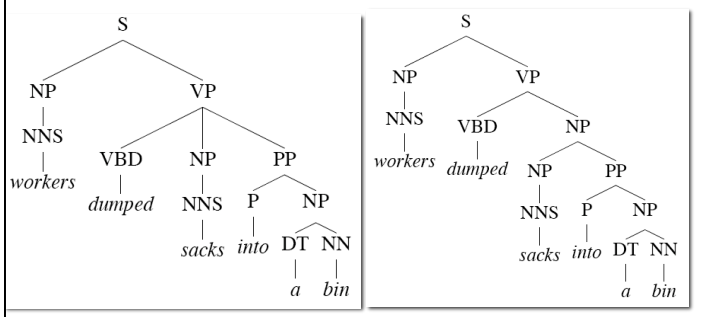
Problems with PCFGs

- The probability model we're using is just based on the the bag of rules in the derivation…
  1. Doesn't take the actual words into account in any useful way.
  2. Doesn't take into account *where* in the derivation a rule is used
  3. *Doesn't work terribly well*
     - That is, the most probable parse isn't usually the right one (the one in the treebank test set).

Specific problems

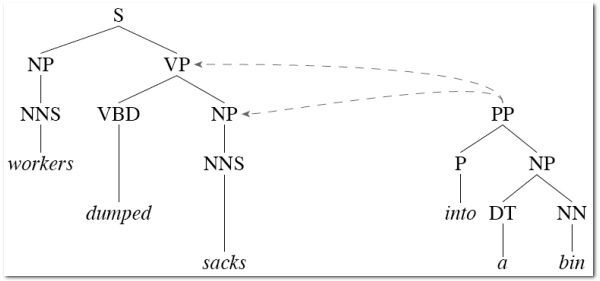
- Attachment ambiguities
  - PP attachment
  - Coordination problems

PP Attachment



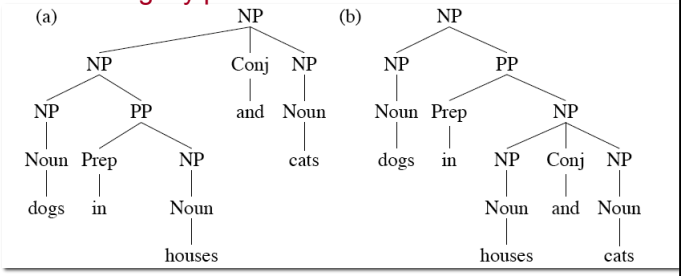
PP Attachment

- Another view.



Coordination

Most grammars have a rule (implicitly) of the form
X -> X and X. This leads to massive ambiguity problems.



Improved Approaches to Statistical Parsing

- We'll look at two approaches to overcoming these shortcomings
  1. Rewrite the grammar to better capture the dependencies among rules
  2. Integrate lexical dependencies into the model
     1. And come up with the independence assumptions needed to make it work.

Solution 1: Rule Rewriting

- The grammar rewriting approach attempts to capture local tree information by rewriting the grammar so that the rules capture the regularities we want.
  - By splitting and merging the non-terminals in the grammar.
    - Example: split NPs into different classes…
  - Remember, we rewrote the grammar rules for CKY, and we rewrote the IOB tags.

Example: NPs

- Our CFG rules for NPs don't condition on where in a tree the rule is applied
- But we know that not all the rules occur with equal frequency in all contexts.
  - Consider *NP*s that involve pronouns vs. those that don't.

	Pronoun	Non-Pronoun
Subject	91%	9%
Object	34%	66%

Other Examples

- There are lots of other examples like this in any treebank
  - Many at the part of speech level
  - Recall that many decisions made in annotation efforts are directed towards improving annotator agreement, not towards doing the right thing.
    - Often this involves conflating distinct classes into a larger class
      - TO, IN, Det, etc.

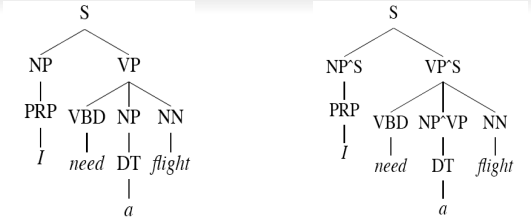
Rule Rewriting

- Three approaches
  - Use linguistic knowledge to directly rewrite rules by hand
    - NP_Obj and the NP_Subj approach
  - Automatically rewrite the rules using context to capture some of what we want
    - Ie. Incorporate context into a context-free approach
  - Search through the space of rewrites for the grammar that maximizes the probability of the training set

Local Context Approach

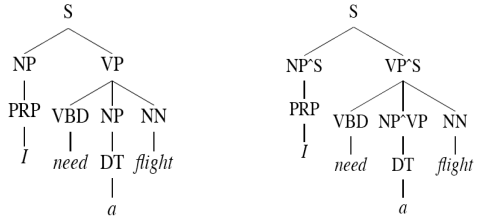
- Condition the rules based on their parent nodes
  - This splitting based on tree-context captures some of the linguistic intuitions

Parent Annotation



- Now we have non-terminals NP^S and NP^VP that should capture the subject/object and pronoun/full NP cases. That is...
  - The rules are now
    - NP^S -> PRP
    - NP^VP -> DT
    - VP^S -> NP^VP

Parent Annotation



- Recall what's going on here. We're in effect rewriting the treebank, thus rewriting the grammar.
- And changing the probabilities since they're being derived from different counts…
  - And if we're splitting what's happening to the counts?

Auto Rewriting

- If this is such a good idea we may as well apply a learning approach to it.
- Start with a grammar (perhaps a treebank grammar)
- Search through the space of splits/merges for the grammar that in some sense maximizes parsing performance on the training/development set.

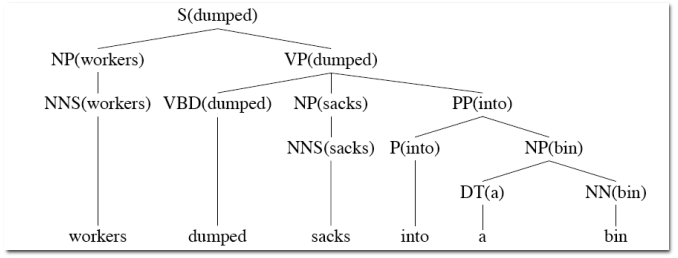
Auto Rewriting

- Basic idea…
  - Split every non-terminal into two new non-terminals across the entire grammar (X becomes X1 and X2).
  - Duplicate all the rules of the grammar that use X, dividing the probability mass of the original rule almost equally.
  - Run EM to readjust the rule probabilities
  - Perform a merge step to back off the splits that look like they don't really do any good.

Solution 2: Lexicalized Grammars

- Lexicalize the grammars with heads
- Compute the rule probabilities on these lexicalized rules
- Run Prob CKY as before

Dumped Example



How?

- We used to have
  - VP -> V NP PP P(rule|VP)
    - That's the count of this rule divided by the number of VPs in a treebank
- Now we have fully lexicalized rules...
  - VP(dumped)-> V(dumped) NP(sacks)PP(into)

P(r|VP ^ dumped is the verb ^ sacks is the head of the NP ^ into is the head of the PP)

To get the counts for that..

Declare Independence

- When stuck, exploit independence and collect the statistics you can…
- There are a larger number of ways to do this...
- Let's consider one generative story: given a rule we'll
  1. Generate the head
  2. Generate the stuff to the left of the head
  3. Generate the stuff to the right of the head

Example

- So the probability of a lexicalized rule such as
  - VP(dumped) → V(dumped)NP(sacks)PP(into)
- Is the product of the probability of
  - *"dumped"* as the head
  - With nothing to its left
  - "*sacks*" as the head of the first right-side thing
  - "*into*" as the head of the next right-side element
  - And nothing after that

Example

- That is, the rule probability for

$$P(VP(dumped,VBD) \rightarrow VBD(dumped, VBD)\ NP(sacks,NNS)\ PP(into,P))$$

is estimated as

$$\begin{aligned} P_H(VBD|VP, dumped) &\times P_L(STOP|VP,VBD,dumped) \\ &\times P_R(NP(sacks,NNS)|VP,VBD,dumped) \\ &\times P_R(PP(into,P)|VP,VBD,dumped) \\ &\times P_R(STOP|VP,VBD,dumped) \end{aligned}$$

Framework

- That's just one simple model
  - Collins Model 1
- You can imagine a gazzillion other assumptions that might lead to better models
- You just have to make sure that you can get the counts you need
- And that it can be used/exploited efficiently during decoding