Today 2/28

• Finish Grammars
  • Treebanks
  • Parsing

Grammars

• Before you can parse you need a grammar.
• So where do grammars come from?
  • Grammar Engineering
    • Lovingly hand-crafted decades-long efforts by humans to write grammars (typically in some particular grammar formalism of interest to the linguists developing the grammar).
  • Treebanks
    • Semi-automatically generated sets of parse trees for the sentences in some corpus. Typically in a generic lowest common denominator formalism (of no particular interest to any modern linguist).
TreeBank Grammars

- Reading off the grammar...
- The grammar is the set of rules (local subtrees) that occur in the annotated corpus
- They tend to avoid recursion (and elegance and parsimony)
  • i.e. they tend to the flat and redundant
- Penn TreeBank (III) has about 17500 grammar rules under this definition.

TreeBanks

TreeBanks

{(S
  (NP-SBJ (DT That)
   (JJ cold) (,) (JJ empty) (NN sky) )
  (VP (VBD was)
   (ADJP-PRED (JJ full)
     (PP (IN of)
       (NP (NN fire)
         (CC and)
         (NP (NN light) )))
     (,. ) )
)}
Sample Rules

NP → DT JJ NNS
NP → DT JJ NN NN
NP → DT JJ CD NNS
NP → RB DT JJ NN NN
NP → RB DT JJ JJ NNS
NP → DT JJ JJ HNP NNS
NP → DT NNP NNP NNP NNP JJ NN
NP → DT JJ NNP CC JJ JJ NN NNS
NP → RB DT JJ JJ NN SBAR
NP → DT VBD JJ HNP NNP CC HNP
NP → DT JJ HNP JJ NNP CC NN NNS NN
NP → DT JJ JJ VBD NN NNP NNP FW NNP
NP → NP JJ JJ "SBAR " NNS

Example

NP → NP JJ , JJ "SBAR " NNS

(11.10) [NP Shearon’s] [JJ easy-to-film], [JJ black-and-white] “[SBAR Where We Stand]” [NNS commercials]

TreeBanks

- TreeBanks provide a grammar (of a sort).
- As we’ll see they also provide the training data for various ML approaches to parsing.
- But they can also provide useful data for more purely linguistic pursuits.
  - You might have a theory about whether or not something can happen in particular language.
  - Or a theory about the contexts in which something can happen.
- Treelbanks can give you the means to explore those theories. If you can formulate the questions in the right way and get the data you need.
You might for example like to grep through a file filled with trees.

```
NP < JJ . VP
(SP (NP (DT tha) (JJ austere) (NN company) (NN dormitory)))
(VP (VBV run))
(IP (IN by) (NP (DT a) (JJ prying) (NN caretaker)))
```

Finally, you should have noted a bit of a circular argument here.

Treebanks provide a grammar because we can read the rules of the grammar out of the treebank.

But how did the trees get in there in the first place? There must have been a grammar theory in there someplace...

Typically, not all of the sentences are hand-annotated by humans.

They're automatically parsed and then hand-corrected.
Parsing

- Parsing with CFGs refers to the task of assigning correct trees to input strings
- Correct here means a tree that covers all and only the elements of the input and has an S at the top
- It doesn’t actually mean that the system can select the correct tree from among all the possible trees

Parsing

- As with everything of interest, parsing involves a search which involves the making of choices
- We’ll start with some basic (meaning bad) methods before moving on to the one or two that you need to know

For Now

- Assume...
  - You have all the words already in some buffer
  - The input isn’t POS tagged
  - We won’t worry about morphological analysis
  - All the words are known
Top-Down Parsing

- Since we’re trying to find trees rooted with an S (Sentences) start with the rules that give us an S.
- Then work your way down from there to the words.

Top Down Space

Bottom-Up Parsing

- Of course, we also want trees that cover the input words. So start with trees that link up with the words in the right way.
- Then work your way up from there.
Of course, in both cases we left out how to keep track of the search space and how to make choices
• Which node to try to expand next
• Which grammar rule to use to expand a node
Top-Down and Bottom-Up

- Top-down
  - Only searches for trees that can be answers (i.e. S’s)
  - But also suggests trees that are not consistent with any of the words
- Bottom-up
  - Only forms trees consistent with the words
  - But suggest trees that make no sense globally

Problems

- Even with the best filtering, backtracking methods are doomed if they don’t address certain problems
  - Ambiguity
  - Shared subproblems

Ambiguity
 Shared Sub-Problems

• No matter what kind of search (top-down or bottom-up or mixed) that we choose.
  • We don’t want to unnecessarily redo work we’ve already done.

 Shared Sub-Problems

• Consider
  • A flight from Indianapolis to Houston on TWA

 Shared Sub-Problems

• Assume a top-down parse making bad initial choices on the Nominal rule.
  • In particular…
    • Nominal -> Nominal Noun
    • Nominal -> Nominal PP
Shared Sub-Problems

NP
  | Det Nominal
  |  a Noun
  |  flight...

Shared Sub-Problems

NP
  | Det Nominal
  |  a Nominal PP
  |  from Indianapolis...
  |  flight

Shared Sub-Problems

NP
  | Det Nominal
  |  a Nominal PP
  |  from Indianapolis...
  |  to Houston...
  |  flight
**Shared Sub-Problems**

![Diagram of shared sub-problems]

---

**Break**

- Next quiz will be pushed back...
- Readings for this section will be from
  - Chapters 12, 13, 14

---

**Parsing**

- We’re going to cover from Chapter 13
  - CKY (today)
  - Earley (Thursday)
- Both are dynamic programming solutions that run in O(n^3) time.
  - CKY is bottom-up
  - Earley is top-down
Dynamic Programming

- DP methods fill tables with partial results and
  - Do not do too much avoidable repeated work
  - Solve exponential problems in polynomial time (sort of)
  - Efficiently store ambiguous structures with shared sub-parts.

CKY Parsing

- First we’ll limit our grammar to epsilon-free, binary rules (more later)
- Consider the rule A -> BC
  - If there is an A in the input then there must be a B followed by a C in the input.
  - If the A spans from i to j in the input then there must be some k st. i<k<j
    - i.e. The B splits from the C someplace.
CKY

• So let’s build a table so that an A spanning from i to j in the input is placed in cell [i,j] in the table.
• So a non-terminal spanning an entire string will sit in cell [0, n]
• If we build the table bottom up we’ll know that the parts of the A must go from i to k and from k to j

• Meaning that for a rule like A -> B C we should look for a B in [i,k] and a C in [k,j].
• In other words, if we think there might be an A spanning i,j in the input… AND
• A -> B C is a rule in the grammar THEN
• There must be a B in [i,k] and a C in [k,j] for some i<k<j

• So to fill the table loop over the cell[i,j] values in some systematic way
  • What constraint should we put on that?
  • For each cell loop over the appropriate k values to search for things to add.
CKY Table

CKY Algorithm

function CKY-Parse(word, grammar) returns table

for j ← 1 to LENGTH(word) do
    table[1..j, 1..j] = {a | a → word[j] ∈ grammar}

for i ← j to 1 do
    for k ← i to j do
        for a ∈ grammar
            for A → BC ∈ grammar
                if table[a, k] and table[B, k] and table[C, k]

CKY Parsing

• Is that really a parser?
Note

• We arranged the loops to fill the table a column at a time, from left to right, bottom to top.
• This assures us that whenever we’re filling a cell, the parts needed to fill it are already in the table (to the left and below)

Example

Other Ways to Do It?

• Are there any other sensible ways to fill the table that still guarantee that the cells we need are already filled?
Other Ways to Do It?

Sample Grammar

Problem

- What if your grammar isn’t binary?
  - As in the case of the TreeBank grammar?
- Convert it to binary… any arbitrary CFG can be rewritten into Chomsky-Normal Form automatically.
- What does this mean?
  - The resulting grammar accepts (and rejects) the same set of strings as the original grammar.
  - But the resulting derivations (trees) are different.
Problem

• More specifically, rules have to be of the form
  A -> B C
Or
  A -> w

That is rules can expand to either 2 non-terminals or to a single terminal.

Binarization Intuition

• Eliminate chains of unit productions.
• Introduce new intermediate non-terminals into the grammar that distribute rules with length > 2 over several rules. So…
  S -> A B C
• Turns into
  S -> X C
  X -> A B

Where X is a symbol that doesn’t occur anywhere else in the grammar.

CNF Conversion
**CKY Algorithm**

Function CKY-PARSE(word, grammar) returns table

for j ← 1 to LENGTH(word) do
  table[0, j] ← { | A → word[0] | ∈ grammar }

for i ← j to 1 do
  for k ← i to j do
    table[i, k] ← \( | A → BC | ∈ grammar \)
    \( B ∈ table[i, j] \)
    \( C ∈ table[j, k] \)

**Example**

```
<table>
<thead>
<tr>
<th>Index</th>
<th>Left</th>
<th>Right</th>
<th>Inside</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Filling column 5

**Example**

```
<table>
<thead>
<tr>
<th>Index</th>
<th>Left</th>
<th>Right</th>
<th>Inside</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```