Identifying Hierarchical Structure in Sequences

Sean Whalen 07.19.06
Review of Grammars
Syntax vs Semantics

- **Syntax** - ‘the pattern of sentences in a language’
- **CS** - ‘the rules governing the formation of statements in a programming language’

- **Semantics** - ‘the study of language meaning’
- **CS** - ‘the meaning of a string in some language, as opposed to syntax which describes how symbols may be combined independent of their meaning’
Digits

• How might we express the syntactic structure of positive integers?
  • digit ::= 0    digit ::= 1    digit ::= 2    ...    digit ::= 9
  • ::= means ‘can be a’ (digit ::= 1 means ‘a digit can be a 1’). We can collect these ‘rules’ together using the or operator
  • digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  • Essentially, digit generates elements in the set {0,1,2,3,4,5,6,7,8,9}
Terminology

• We gave 10 rules (called productions) describing possible digits

• Left hand side of a production is called a nonterminal, because it can generate additional symbols

• Right hand side may contain terminals, more nonterminals, or both

• Replacing digit with something which ‘can be a digit’ is called a derivation

• Given a production, what set of strings can be derived (generated)
Formalization

• A grammar can be defined as \( G = (N, T, P, S) \)
  
  • \( N \) - a set of nonterminals
  • \( T \) - a set of terminals
  • \( P \) - a set of productions
  • \( S \) - a start symbol
Numbers

• How would we generate a number with multiple digits?
  • number ::= digit digit
  • number ::= digit digit digit

• Can we use a finite number of productions?

• Recursion!
  • number ::= number digit | digit
production  \rightarrow \text{number ::= number digit} \mid \text{digit}

production  \rightarrow \text{digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}

\text{can be a nonterminal nonterminal}

\text{nonterminal nonterminal or}

\text{terminal \ldots terminal}
• Let’s say a name in the phonebook is defined as last name, first name middle initial

  name ::= last_name , first_name middle_initial

• Middle initial is easiest

  char ::= A | B | C ... | Z

  middle_initial ::= char
Names

- Similar to number, we need a recursive definition for a string of characters
  - char_list ::= char_list char | char

- Now we can define names
  - first_name ::= char_list
  - last_name ::= char_list
Database Record

- One can describe the structure of a database entry this way
  - student_record ::= name address ssn
  - address ::= house_address | apt_address
  - house_address ::= number street city state zip
  - apt_address ::= number street apt_num city state zip
- This is a Backus-Naur Form (BNF) context free grammar (CFG)
Precedence

- assignment ::= identifier = expression ;
- expression ::= term | expression + term | expression - term
- term ::= factor | term * factor | term / factor
- factor ::= identifier | literal | (expression)
Grammatical Inference
Grammatical Inference

- The task of learning (inferring) a grammar from a set of sample strings

- Automata / grammar equivalence
  - For any regular grammar $G$ there is a finite automaton accepting $L(G)$
  - Same for context free grammars / pushdown automata
  - Thus, learning automata from sample strings is also GI

- Probabilistic versions - learn a FSM that generates strings for some PD
Algorithms

• Regular Positive Negative Inference (RPNI, Oncina & Garcia 92) - Given labeled samples both in (positive) and not in (negative) the target language, infer the minimal DFA acceptor

• Alergia (Carrasco & Oncina 94/99) - Given a set of example strings, infer a stochastic context free grammar

• MDI (DuPont et al 2000) - Refinement of Alergia, addresses global control of generalization from training sample using KL divergence
Sequitur
Nevill-Manning and Witten, Journal of AI Research 97

• ‘Sequitur is an algorithm that infers a hierarchical structure from a sequence of discrete symbols by replacing repeated phrases with a grammatical rule that generates the phrase, and continues this process recursively.’

• Creates a set of productions and a start symbol which can derive original string

• The grammar represents hierarchical structure, and simultaneously performs lossless compression (similar to Lempel-Ziv)
High-Level Algorithm

- Scan input string, adding one symbol at a time to starting production
- For each **digram** (2 consecutive letters) formed by last 2 symbols
  - Store digram in a table
  - If it’s already in the table, it’s been seen before
    - Create production - replace current and matched digram with nonterminal for new production
Constraints

• No pair of adjacent symbols can appear more than once in the grammar (*digram uniqueness*)
  • This includes nonterminals for productions
  • Digrams containing nonterminals create the hierarchy
• Every rule is used more than once (*rule utility*)
  • Delete rules when used only once
Example 1

- string > abcdbc
- Parse consecutive digrams and store
  - Store ab, store bc, store cd, store bc
  - bc matches! create new rule A ::= bc
- S ::= aAdA
- A ::= bc
Example 2 - Nested

- string > abcdbcabcdbc
- S ::= AA
- A ::= aBdB
- B ::= bc
Example 3 - Violation

- string > abcdbcabcdbc (same)
- S ::= AA
- A ::= abcdbc
- bc occurs twice in rule A
### Step by Step

<table>
<thead>
<tr>
<th>symbol number</th>
<th>the string so far</th>
<th>resulting grammar</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>$S \rightarrow a$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ab</td>
<td>$S \rightarrow ab$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>abc</td>
<td>$S \rightarrow abc$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>abcd</td>
<td>$S \rightarrow abcd$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>abcdab</td>
<td>$S \rightarrow abcdab$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>abcdabc</td>
<td>$S \rightarrow abcdabc$</td>
<td>$bc$ appears twice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S \rightarrow aAdA$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A \rightarrow bc$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>abcdabcab</td>
<td>$S \rightarrow aAdAab$</td>
<td>$bc$ appears twice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A \rightarrow bc$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>abcdabcabc</td>
<td>$S \rightarrow aAdAabc$</td>
<td>$bc$ appears twice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A \rightarrow bc$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S \rightarrow aAdAaaA$</td>
<td>$bc$ appears twice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A \rightarrow bc$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S \rightarrow BdAB$</td>
<td>$bc$ appears twice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$A \rightarrow bc$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$B \rightarrow aA$</td>
<td></td>
</tr>
</tbody>
</table>
Implementation Issues

- Appending a symbol to start symbol S
- Using an existing rule - substitute nonterminal for two symbols
- Creating a new rule - new nonterminal, insert 2 symbols, substitute
- Deleting a rule - replace nonterminal with contents
- Data structures: doubly linked list (symbols), hash table (digrams)
# Digram Table

<table>
<thead>
<tr>
<th>Action</th>
<th>grammar</th>
<th>change in digrams</th>
<th>digram index</th>
</tr>
</thead>
<tbody>
<tr>
<td>observe symbol 'c'</td>
<td>$S \to abcdbc$</td>
<td></td>
<td>${ab, bc, cd, db}$</td>
</tr>
<tr>
<td>make new rule A</td>
<td>$S \to abdcbd$</td>
<td>$bc$ updated</td>
<td>${ab, bc, cd, db}$</td>
</tr>
<tr>
<td>substitute A for bc</td>
<td>$S \to aAdbc$</td>
<td>$ab, cd$ removed,</td>
<td>${bc, db, aA, Ad}$</td>
</tr>
<tr>
<td></td>
<td>$A \to bc$</td>
<td>$aA, Ad$ added</td>
<td></td>
</tr>
<tr>
<td>substitute A for bc</td>
<td>$S \to aAdA$</td>
<td>$db$ removed,</td>
<td>${bc, dA, aA, Ad}$</td>
</tr>
<tr>
<td></td>
<td>$A \to bc$</td>
<td>$dA$ added</td>
<td></td>
</tr>
</tbody>
</table>
Complexity

• Linear time in length of string (good), linear space in size of grammar (bad)

• Creation/deletion/substitution/lookup each have constant number of operations due to doubly linked list and hash table

• Number of rules can increase without bound for unstructured data

• Linearity proof is valid for $10^9$ symbols on 32-bit machines
Performance

Graph 1: Rules in grammar vs. input symbols

Graph 2: Time (seconds) vs. input symbols
Compression Performance

Compression Quality

• Neville-Manning and Witten 1997

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>comp</th>
<th>gzip</th>
<th>sequitur</th>
<th>PPMC</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111261</td>
<td>3.35</td>
<td>2.51</td>
<td>2.46</td>
<td>2.12</td>
<td>1.98</td>
</tr>
<tr>
<td>book</td>
<td>766771</td>
<td>3.46</td>
<td>3.35</td>
<td>2.82</td>
<td>2.52</td>
<td>2.42</td>
</tr>
<tr>
<td>geo</td>
<td>102400</td>
<td>6.08</td>
<td>5.34</td>
<td>4.74</td>
<td>5.01</td>
<td>4.45</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>4.17</td>
<td>2.63</td>
<td>2.68</td>
<td>2.77</td>
<td>2.48</td>
</tr>
<tr>
<td>pic</td>
<td>513216</td>
<td>0.97</td>
<td>0.82</td>
<td>0.90</td>
<td>0.98</td>
<td>0.78</td>
</tr>
<tr>
<td>progc</td>
<td>38611</td>
<td>3.87</td>
<td>2.68</td>
<td>2.83</td>
<td>2.49</td>
<td>2.53</td>
</tr>
</tbody>
</table>

Files from the Calgary Corpus
Units in bits per character (3 bits)
compress - based on LZW
gzip - based on LZ77
PPMC - adaptive arithmetic coding with context
bzip2 - Burrows-Wheeler block sorting
Data

please porridge hot,
please porridge cold,
please porridge in the pot,
 nine days old.
some like it hot,
some like it cold,
some like it in the pot,
in nine days old.

Grammar

Run sequitur

Source code:

sequitur
rule
symbol
nonTerminal
terminal
guard
Applet sequitur started
Application

• Run algorithm on protocol data
• Convert resulting grammar to DFA
• How big is the resulting model?
• Is the structure of the protocol more or less apparent?