Regular expressions and automata

Introduction
Finite State Automaton (FSA)
Finite State Transducers (FST)
Regular expressions

Standard notation for characterizing text sequences

Specifying text strings:

• Web search: `woodchuck` (with an optional final `s`) (lower/upper case)
• Computation of frequencies
• Word-processing (Word, Emacs, Perl)
Regular expressions (REs)

A RE formula is a special language (an algebraic notation) to specify simple classes of strings: a sequence of symbols (i.e., alphanumeric characters).

woodchucks, a, song,!, Mary says

REs are used to

- Specify search strings - to define a pattern to search through a corpus
- Define a language
Regular expressions

• Basically they are combinations of simple units (character or strings) with connectives as concatenation, disjunction, option, kleene star, etc.

• Used in languages as Perl or Python and Unix commands as grep, replace,...
Regular expressions and automata

- Regular expressions can be implemented by the finite-state automaton.
- Finite State Automaton (FSA) a significant tool of computational linguistics. They are related to other computational tools:
  - Finite State Transducers (FST)
  - N-gram
  - Hidden Markov Models
Regular expressions (REs)

- Case sensitive: woodchucks different from Woodchucks
- [] means disjunction
  [Ww]oodchucks
  [1234567890] (any digit)
  [A-Z] an uppercase letter
- [^] means cannot be
  [^A-Z] not an uppercase letter
  [^Ss] neither 'S' nor 's'
Regular expressions

- ? means preceding character or nothing

**Woodchucks?** means Woodchucks or Woodchuck

**color?r** color or colour

- * (kleene star) - zero or more occurrences of the immediately previous character

  a* any string or zero or more as (a, aa, hello)

- [0-9][0-9]* - any integer

- + one or more occurrences

[0-9]+
Regular expressions

- **Disjunction operator** |  cat|dog
- There are other more complex operators
- Operator precedence hierarchy
- Very useful in substitutions (i.e. Dialogue)
Regular expressions
Useful to write patterns:
Examples of substitutions in dialogue

User: Men are all alike
ELIZA: IN WHAT WAY

s/.*/all.*/ IN WHAT WAY

User: They're always bugging us about something
ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE

s/*always.*/ CAN YOU THINK OF A SPECIFIC EXAMPLE
- Acronym detection

patterns acrophile

acro1 = re.compile('^(\[A-Z][,\.-/_\])+$')
acro2 = re.compile('^[A-Z]+$')
acro3 = re.compile('^[\d]*[A-Z](\d[A-Z]+)$')
acro4 = re.compile('^[A-Z][A-Z][A-Z]+[A-Za-z]+$')
acro5 = re.compile('^[A-Z][A-Z]+[A-Za-z][A-Z]+$')
acro6 = re.compile('^[A-Z][,\.-/_\]{2,9}(s|s)?$')
acro7 = re.compile('^[A-Z]{2,9}(s|s)?$')
acro8 = re.compile('^[A-Z]*[\d][-_]?[A-Z]+$')
acro9 = re.compile('^[A-Z]+[A-Za-z]+[A-Z]+$')
Some readings


Roche and Schabes 1997

References to Finite-State Methods in Natural Language Processing
http://www.cis.upenn.edu/~cis639/docs/fs.refs.html
Some toolbox

ATT FSM tools
http://www2.research.att.com/~fsmtools/fsm/

Beesley, Kartunnen book
http://www.stanford.edu/~laurik/fsmbook/home.html

Carmel
http://www.isi.edu/licensed-sw/carmel/

Dan Colish's PyFSA (Python FSA)
https://github.com/dcolish/PyFSA
Equivalence

Regular Expressions
Regular Languages
Finite State Automaton
Regular Languages (RL)

Alphabet (vocabulary) $\Sigma$
Concatenation operation $\Sigma^*$ strings over $\Sigma$ (free monoid)
Language $L \subseteq \Sigma^*$
Languages and grammars
L, L₁ y L₂ are languages

operations

concatenation

\[ L_1 \cdot L_2 = \{ u \cdot v | u \in L_1 \land v \in L_2 \} \]

union

\[ L_1 \cup L_2 = \{ u | u \in L_1 \lor u \in L_2 \} \]

intersection

\[ L_1 \cap L_2 = \{ u | u \in L_1 \land u \in L_2 \} \]

difference

\[ L_1 - L_2 = \{ u | u \in L_1 \land u \notin L_2 \} \]

complement

\[ \overline{L} = \Sigma - L \]
Finite State Automata (FSA)

\[<\Sigma, Q, i, F, E>\]

- \(\Sigma\): alphabet
- \(Q\): finite set of states
- \(i \in Q\): initial state
- \(F \subseteq Q\): final states set
- \(E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q\): arc set
- \(E: \{d \mid d: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q\}\): transitions set
Example 1: Recognizes multiple of 2 codified in binary

Examples of numbers recognized
0
10 (2 in decimal)
100 (4 in decimal)
110 (6 in decimal)

State 0:
The string recognized till now ends with 0

State 1:
The string recognized till now ends with 1
Example 2: Recognizes multiple of 3 codified in binary

**State 0:** The string recognized till now is multiple of 3
**State 1:** The string recognized till now is multiple of 3 + 1
**State 2:** The string recognized till now is multiple of 3 + 2

The transition from a state to the following multiplies by 2 the current string and adds to it the current tag
Tabular representation of the FSA

Recognizes multiple of 3 codified in binary

NLP FS Models
Properties of regular languages (RL) and FSA

Let $A$ a FSA

$L(A)$ is the language generated (recognized) by $A$

The class of RL (or FSA) is closed under

- union
- intersection
- concatenation
- complement
- Kleene star ($A^*$)

FSA can be determined

FSA can be minimized
The following properties of FSA are decidable

\( w \in L(A) ? \)
\( L(A) = \emptyset ? \)
\( L(A) = \Sigma^* ? \)
\( L(A_1) \subseteq L(A_2) ? \)
\( L(A_1) = L(A_2) ? \)

Only the first two are for context free grammars (CFG), the most used grammars
Example of the use of closure properties

Representation of the Lexicon

Let S the FSA: Representation of the sentence with POS tags
We are interested on

\[ S - (\Sigma^* \cdot C1 \cdot \Sigma^*) - (\Sigma^* \cdot C2 \cdot \Sigma^*) = S - (\Sigma^* \cdot (C1 \cup C2) \cdot \Sigma^*) \]
From the union of negative rules we can build a Negative grammar $G = \Sigma^* \cdot (C_1 \cup C_2 \cup \ldots \cup C_n) \cdot \Sigma^*$

Diagram:

- Det this
- Det that
- Pro
- V
- N
The difference between the two FSA S-G will result on:

Most of the ambiguities have been solved
Finite State Transducers (FST)

\(<\Sigma_1, \Sigma_2, Q, i, F, E>\)

- \(\Sigma_1\): input alphabet
- \(\Sigma_2\): output alphabet
- frequently \(\Sigma_1 = \Sigma_2 = \Sigma\)
- \(Q\): finite states set
- \(i \in Q\): initial state
- \(F \subseteq Q\): final states set
- \(E \subseteq Q \times (\Sigma_1^* \times \Sigma_2^*) \times Q\): arcs set

NLP FS Models
Example 3

Td3: division by 3 of a binary string

$\Sigma_1 = \Sigma_2 = \Sigma = \{0,1\}$
Example 3

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>110</td>
<td>010</td>
</tr>
<tr>
<td>1001</td>
<td>0011</td>
</tr>
<tr>
<td>1100</td>
<td>0100</td>
</tr>
<tr>
<td>1111</td>
<td>0101</td>
</tr>
<tr>
<td>10010</td>
<td>00110</td>
</tr>
</tbody>
</table>

Td3: division by 3 of a binary string

$\Sigma_1 = \Sigma_2 = \Sigma = \{0, 1\}$
State 0:
Recognized: 3k
Emitted: k

State 1:
Recognized: 3k+1
Emitted: k

State 2:
Recognized: 3k+2
Emitted: k

invariant:
emitted * 3 = Recognized

invariant:
emitted * 3 + 1 = Recognized

invariant:
emitted * 3 + 2 = Recognized

NLP FS Models
state 0:
Recognized: $3k$
Emitted: $k$

State 0 satisfies invariant

State 1

State 1 satisfies invariant
State 0 satisfies invariant

 consums: 1
 emits: 1
 recognized: \((3k+1) \times 2 + 1 = 6k + 3\)
 emitted: \(k \times 2 + 1 = 2k + 1\)

State 2 satisfies invariant

 consums: 0
 emits: 0
 recognized: \((3k+1) \times 2 = 6k + 2\)
 Emited: \(k \times 2 = 2k\)

State 1:

 recognized: \(3k + 1\)
 emitted: \(k\)

NLP FS Models
State 1 satisfies invariant

State 2 satisfies invariant

state 2:
recognized: $3k+2$
emited: $k$

consums: 0
emits: 1
recognized: $(3k+2)*2 = 6k + 4$
emited: $k*2 + 1 = 2k + 1$

consums: 1
emits: 1
recognized: $(3k+2)*2 + 1 = 6k + 5$
emited: $k*2 + 1 = 2k + 1$
FSA associated with a FST

FST $<\Sigma_1, \Sigma_2, Q, i, F, E>$

FSA $<\Sigma, Q, i, F, E'>$

$\Sigma = \Sigma_1 \times \Sigma_2$

$(q_1, (a, b), q_2) \in E' \iff (q_1, a, b, q_2) \in E$
FST

Projections of a FST

FST  \( T = \langle \Sigma_1, \Sigma_2, Q, i, F, E \rangle \)

First projection

\( P_1(T) \langle \Sigma_1, Q, i, F, E_{P_1} \rangle \)
\( E_{P_1} = \{(q,a,q') \mid (q,a,b,q') \in E\} \)

Second projection

\( P_2(T) \langle \Sigma_2, Q, i, F, E_{P_2} \rangle \)
\( E_{P_2} = \{(q,b,q') \mid (q,a,b,q') \in E\} \)
FST are closed under
  union
  inversion
    example: \( Td3^{-1} \) is equivalent to multiply by 3
  composition
    example: \( Td9 = Td3 \circ Td3 \)
FST are **not** closed under intersection
Traverse the FST in all forms compatible with the input (using backtracking if needed) until reaching a final state and generate the corresponding output.

Consider input as a FSA and compute the intersection of the FSA and the FST.
Not all FST are determinizable, if it is the case they are named **subsequential**

The **non deterministic** FST is equivalent to the **deterministic** one
Aplications of FSA(and FST)

Increasing use in NLP

  - Morphology
  - Phonology
  - Lexical generation
  - ASR (Automatic Speech Recognition)
  - POS tagging
  - Simplification of Grammars
  - Information Extraction
• Why FSA (and FST)?
  • Temporal and spatial efficiency
  • Some FSA can be determined and optimized for leading to more compact representations
  • Possibility to be used in cascade form