Finite state morphology and phonology

Natural Language Processing
LING/CSCI 5832

Mans Hulden
Dept. of Linguistics
mans.hulden@colorado.edu

Jan 20 2014
FSMs for practical NLP tasks

(1) How FSMs are used in modeling sound systems (phonology)

(2) For modeling word-formation

(3) Derivative products of the above (spell checkers, lemmatizers, grammar checkers, components of larger systems)
Plan

(1) Recap finite automata and transducers + basic algorithms

(2) Look at an extended calculus for manipulating FSMs (automata + transducers) suitable for NLP

(3) See how these are used in natural language applications
Recap: anatomy of a FSA

Regular expression

\[ L = a \, b^* \, c \]

Graph representation

Formal definition

\[ Q = \{0,1,2\} \text{ (set of states)} \]
\[ \Sigma = \{a,b,c\} \text{ (alphabet)} \]
\[ q_0 = 0 \text{ (initial state)} \]
\[ F = \{2\} \text{ (set of final states)} \]
\[ \delta(0,a) = 1, \delta(1,b) = 1, \delta(1,c) = 2 \text{ (transition function)} \]
Recap: anatomy of a FSA

Regular expression

\[ L = a \ b^* \ c \]

Interpretation

• An FSA defines a set of strings

• In this case \( L = \{ac, abc, abbc, \ldots\} \)

• These sets are the regular sets
Recap: Kleene’s Theorem

A language is regular iff it is accepted by some FA.

Proof is constructive: can convert between representations.

\[(a|b^* c)^* a b a^* | (a b^* a | a a^*)\]
## Recap: Kleene’s Theorem

### Kleene’s Theorem: \( \text{regexp} \rightarrow \text{FA} \)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
<th>FSM construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \varepsilon )</td>
<td>The empty string</td>
<td><img src="empty-string.png" alt="Diagram" /></td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>The empty language</td>
<td><img src="empty-language.png" alt="Diagram" /></td>
</tr>
<tr>
<td>( a )</td>
<td>A single symbol</td>
<td><img src="single-symbol.png" alt="Diagram" /></td>
</tr>
<tr>
<td>( A^* )</td>
<td>Kleene star of a language</td>
<td><img src="kleene-star.png" alt="Diagram" /></td>
</tr>
<tr>
<td>( AB )</td>
<td>Concatenation of two languages</td>
<td><img src="concatenation.png" alt="Diagram" /></td>
</tr>
<tr>
<td>( A</td>
<td>B )</td>
<td>Union of two languages</td>
</tr>
</tbody>
</table>

### FA \( \rightarrow \) regexp done with “state elimination algorithm” (easier, but let’s skip it)
The Thompson construction

$(a|b)^*$

Diagram:

- A circle labeled $a$ points to a circle labeled $a$.
- A circle labeled $b$ points to a circle labeled $b$. 
The Thompson construction

\[(a|b)^*\]

(a|b)
The Thompson construction

$$(a|b)^*$$

Diagram:

- Initial state with transitions to states labeled $a$ and $b$.
- States connected by transitions labeled $a$, $b$, $\epsilon$, and loops on each state.

Graphical representation of $$(a|b)^*$$
The Thompson construction

\[(a|b)^*\]

determinization & minimization algorithm
Recap: Kleene’s Theorem

• Kleene’s Theorem only uses one Boolean operation on sets, union

• But FSA are closed under other set operations: complement, intersection, set subtraction

• It’s difficult to appreciate the power of finite-state models without a richer calculus...

• In fact, the most fruitful approach is to forget about states and transitions and tapes and reason in terms of sets and relations
What language does the FSA represent?

Σ = \{a, b, c\}
Reasoning about automata

Automaton

\[ \Sigma = \{a, b, c\} \]

Equivalent regular expression with \{\,, \|, *, \}:

\[(b|c|a*a*c)*a*a*b(aa*b|(b|aa*c)(b|c|aa*c)*a*a*b)|(b|c)* a((a|ba)|(c|bb)(b|c)*a)|(b|c|a(a|ba)*(c|bb))*\]
Reasoning about automata

Automaton

\[ \Sigma = \{a,b,c\} \]

Equivalent regular expression with \{|,., *\}:

\[
(b|c| aa* c)* aa* b(aa* b|(b| aa* c)(b| c| aa* c)* aa* b)* (b| c)* a((a|ba)|(c|bb)(b| c)* a)* (b| c| a(a|ba)*(c|bb))*
\]

Equivalent regular expression with \{.,¬,\,*\}:

\[
\neg(\Sigma* abc \Sigma*)
\]
Reasoning about automata

\[ \Sigma = \{a, b, c\} \]

Equivalent regular expression with \{|, , \ast\}:
\[(b|a^*c)^*aa^*b(aa^*b|(b|aa^*c)(b|c|aa^*c)^*aa^*b)^*|(b|c)^* a((a|ba)|(c|bb)(b|c)^*a)^*|(b|c|a(a|ba)^*(c|bb))^*\]

Equivalent regular expression with \{|, , \neg\}:
\[\neg(\Sigma^*abc\Sigma^*)\]
not “contains abc”
Reasoning about automata

From “Regular models of phonological rule systems”

The common data structures that our programs manipulate are clearly states, transitions, labels, and label pairs—the building blocks of finite automata and transducers. But many of our initial mistakes and failures arose from attempting also to think in terms of these objects. The automata required to implement even the simplest examples are large and involve considerable subtlety for their construction. To view them from the perspective of states and transitions is much like predicting weather patterns by studying the movements of atoms and molecules or inverting a matrix with a Turing machine. The only hope of success in this domain lies in developing an appropriate set of high-level algebraic operators for reasoning about languages and relations and for justifying a corresponding set of operators and automata for computation.

(Kaplan and Kay, 1994, p.376)
Toward “high-level” algebraic operators

• Add Booleans to regular expression calculus: at least complement (¬), intersection (∩), set subtraction (-))

• Add “useful” operators/shortcuts, e.g.
  - contains(X) = (Σ* X Σ*)

• Example: the language that fulfills the constraint: “i before e except after c”
  ¬contains(cie) & ¬(¬(Σ*c)ei)
The product construction

\[ L_1 = a \ b^* \ c \]

\[ L_2 = a \ b \ c^* \]

\[ L_3 = L_1 \cap L_2 \]
The product construction

\( L_1 = a b^* c \)

\( L_2 = a b c^* \)

\( L_3 = L_1 \cap L_2 \)
The product construction

\[ L_1 = a \, b^* \, c \]

\[ L_2 = a \, b \, c^* \]

\[ L_3 = L_1 \cap L_2 \]
The product construction

\[ L_1 = a \, b^* \, c \]

\[ L_2 = a \, b \, c^* \]

\[ L_3 = L_1 \cap L_2 \]
The product construction

\[ L_1 = a \, b^* \, c \]

\[ L_2 = a \, b \, c^* \]

\[ L_3 = L_1 \cap L_2 \]
The product construction

**Algorithm 3.2: PRODUCTCONSTRUCTION**

**Input:** $FSM_1 = (Q_1, \Sigma, \delta_1, s_0, F_1), FSM_2 = (Q_2, \Sigma, \delta_2, t_0, F_2), OP \in \{\cup, \cap, -\}$

**Output:** $FSM_3 = (Q_3, \Sigma, \delta_3, u_0, F_3)$

1 begin
2     Agenda $\leftarrow (s_0, t_0)$
3     $Q_3 \leftarrow (s_0, t_0)$
4     $u_0 \leftarrow (s_0, t_0)$
5     index $(s_0, t_0)$
6     while Agenda $\neq \emptyset$ do
7         Choose a state pair $(p, q)$ from Agenda
8         foreach pair of transitions $\delta_1(p, x, p') \delta_2(q, x, q')$ do
9             Add $\delta_3((p, q), x, (p', q'))$
10            if $(p', q')$ is not indexed then
11                Index $(p', q')$ and add to Agenda and $Q_3$
12            end
13         end
14     end
15     foreach State $s$ in $Q_3 = (p, q)$ do
16         Add $s$ to $F_3$ iff $p \in F_1$ OP $q \in F_2$
17     end
18 end

In the algorithm, it is assumed that in a transition specified as $\delta_1(p, x, p')$, $x$ refers to any symbol pair occurring on a transition in a transducer. The input arguments must be $\varepsilon$-free: i.e. $\varepsilon$-transitions must not be present.

The way algorithm 3.2 specifies the product construction assumes implicitly that each automaton is complete for the set of possible symbol pairs. In practice we will want to deal with trim automata, and so do not have access to complete transition functions. For the intersection operation this is irrelevant since if there does not exist a common transition for $FSM_1$ and $FSM_2$ and some state pair $(p, q)$, such a transition should not exist in the
Finite state transducers
Recap: anatomy of an FST

### Formal definition

- **Set of states** ($Q$) = \{0, 1, 2, 3\}
- **Alphabet** ($\Sigma$) = \{a, b, c, d\}
- **Initial state** ($q_0$) = 0
- **Set of final states** ($F$) = \{0, 1, 2\}
- **Transition function** ($\delta$)

### Graph representation
Recap: anatomy of an FST

Graph representation

Interpretation

• An FST defines a set of string pairs (a relation)

• In this case $T=\{(a,a),(b,b),(c,c), (cad,cdb),...\}$

• These sets are the regular relations

• Trivially bidirectional devices
Algebraic operations on transducers

T U (concatenation)

T | U (union)

T* (Kleene closure)

rev(T) (reversal)

L_1 x L_2 (cross-product)

T o U (composition)
Algebraic operations on transducers

- $T \cup U$ (concatenation)
- $T \mid U$ (union)
- $T^*$ (Kleene closure)
- $\text{rev}(T)$ (reversal)
- $L_1 \times L_2$ (cross-product)
- $T \circ U$ (composition)

Cross-product

Regular languages

$(ab|ac) \times (c|d)$

Cross-product:

$ab \times c$

$ac \times d$
Algebraic operations on transducers

T U (concatenation)
T | U (union)
T* (Kleene closure)
rev(T) (reversal)
L_1 \times L_2 (cross-product)
T \circ U (composition)
Composition: product construction

\[ T_3 = T_1 \circ T_2 \]
**String rewriting operators**

\[ A \rightarrow B / C _ D \]

“Rewrite strings A as B when occurring between C and D”

Example: \((a|e|i|o|u) \rightarrow 0 / _ #\)

delete vowels at the ends of words

Difficult to implement correctly in the general case
Actual single Finnish word (not a compound!) ‘perhaps even because of his/her/it not having an ability to not generalize herself/himself/itself’ (maybe)

Grammatically correct, semantics is elusive, akin to ‘colorless green ideas sleep furiously’

Highly agglutinative languages like this have an astronomical number of “possible words”, even without considering neologisms
Linguistics: a model of word production

epäjärjestelmälistyttämättömyydelläänsäkäänköhän

Modeled by a step-by-step generative process:

‘un’+‘system’ +‘ize’
epä+järjestelmä+lis+...

Modeled by a step-by-step generative process:

epäjärjestelmälistyttämättömyydelläänsäkäänköhän

Modeled by a step-by-step generative process:

epäjärjestelmälistyttämättömyydelläänsäkäänköhän

epäjärjestelmälistyttämättömyydelläänsäkäänköhän
“Generative” word model

in+possible+ity

(1) Pick morphemes from lexicon in right order and combinations (dictated by morphotactics)
“Generative” word model

in+possible+ity

change n to m before p
(nasal assimilation)

im+possible+ity

(1) Pick morphemes from lexicon in right order and combinations (dictated by morphotactics)

(2) Apply sound change rules + orthographic rules
“Generative” word model

1. Pick morphemes from lexicon in right order and combinations (dictated by morphotactics)

   - in+possible+ity
     - change n to m before p (nasal assimilation)
   - im+possible+ity
   - ble+ity > bility
   - im+possibility
   - im+possibility

2. Apply sound change rules + orthographic rules

   - remove boundaries
   - impossibility
Four tricks to model this

(1) Extended operators (Booleans, replacements)

(2) Use alphabet independent algorithms

\[ \Sigma^* \ a \ \Sigma^* \longrightarrow 0 \ \xrightarrow{a} \ 1 \]

(3) Treat automata as “repeating transducers” (“everything is a transducer”)

(4) Model lexicon as an FST (which may just repeat words)
“Generative” word model

Lexicon + morphology

1. Pick morphemes from lexicon in right order and combinations (dictated by morphotactics)

2. Apply sound change rules + orthographic rules

- in+possible+ity
- change n to m before p (nasal assimilation)
- im+possible+ity
- ble+ity → bility
- im+possibility
- remove boundaries
- impossibility
“Generative” word model

Lexicon + morphology

in+possible+ity

n → m / _ + p

im+possible+ity

ble+ity → bility

im+possibility

+ → 0

impossibility

(1) Pick morphemes from lexicon in right order and combinations (dictated by morphotactics)

(2) Apply sound change rules + orthographic rules
(1) Pick morphemes from lexicon in right order and combinations (dictated by morphotactics)

(2) Apply sound change rules + orthographical rules

...then compose
Composition

im+possible+ity

in+possible+ity

im+possible+ity

im+possibility

impossibility
Adding grammatical information

We’d like to be able to get parses with grammatical information:

impossibilities => NEG+possible+ity+NOUN+PLURAL
vs.
   in+possible+ity+s
We’d like to be able to get parses with grammatical information:

impossibilities => NEG+possible+ity+NOUN+PLURAL 

vs.

in+possible+ity+s

Solution: make lexicon a transduction:

IN: NEG+possible+ity+NOUN+PLURAL

OUT: in+possible+ity+s
Composition

NEG+possible+ity+NOUN+PLURAL

in+possible+ity+s

im+possible+ity+s

im+possibility+s

impossibilities
Composition

NEG+possible+ity+NOUN+PLURAL

in+possible+ity+s

im+possible+ity+s

im+possibility+s

impossibilities

impossibility
Compilers

Several finite-state compilers available to do the hard work

- Xerox xfst (http://www.fsmbook.com)
- SFST (https://code.google.com/p/cistern/wiki/SFST)
- HFST (http://hfst.sf.net)
- OpenFST (http://www.openfst.org)
- Foma (http://foma.googlecode.com)

Demo with foma