Finite state morphology and phonology

Natural Language Processing
CSCI 5832

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

Oct 13 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
\[ L = a \, b^* \, c \]

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

Regular expression

$L = a \ b^* \ c$

Interpretation

• An FSA defines a set of strings
  • In this case $L = \{ac, abc, abbc, \ldots\}$

Graph representation

• 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: regexp $\rightarrow$ FA

<table>
<thead>
<tr>
<th>Expression</th>
<th>Definition</th>
<th>FSM construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>The empty string</td>
<td>$\text{ }$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>The empty language</td>
<td>$\text{ }$</td>
</tr>
<tr>
<td>$a$</td>
<td>A single symbol</td>
<td>$\text{ }$</td>
</tr>
<tr>
<td>$A^*$</td>
<td>Kleene star of a language</td>
<td>$\text{ }$</td>
</tr>
<tr>
<td>$AB$</td>
<td>Concatenation of two languages</td>
<td>$\text{ }$</td>
</tr>
<tr>
<td>$A \mid B$</td>
<td>Union of two languages</td>
<td>$\text{ }$</td>
</tr>
</tbody>
</table>

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

\[(a|b)^*\]
The Thompson construction

\[(a|b)^*\]

\[
\begin{array}{c}
\text{a} \\
\rightarrow \\
\text{circle} \\
\end{array}
\]

\[
\begin{array}{c}
\text{b} \\
\rightarrow \\
\text{circle} \\
\end{array}
\]
The Thompson construction

\[(a|b)^*\]
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
Reasoning about automata

What language does the FSA represent?

\[ \Sigma = \{a, b, c\} \]
Reasoning about automata

Automaton

Equivalent regular expression with \{\mid,\cdot,*\}

\[
(b\mid c\mid aa^*c)^*aa^*b(aa^*b\mid(b\mid aa^*c)(b\mid c\mid aa^*c)^*aa^*b)^\ast(b\mid c)^\ast a((a\mid ba)(c\mid bb)(b\mid c)^*a)^\ast(b\mid c\mid a(a\mid ba)^*(c\mid bb))^\ast
\]
Reasoning about automata

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

Equivalent regular expression with \{\|,\cdot,\ast\}:

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

Equivalent regular expression with \{\cdot,\neg,\ast\}:

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

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

Automaton

Equivalent regular expression with \{\mid, \cdot, \ast\}

\[(b\mid c\mid a\ast c)^\ast a\ast b(aa\ast b\mid (b\mid a\ast c)(b\mid c\mid a\ast c)^\ast a\ast b)\ast (b\mid c)^\ast a((a\mid ba)(c\mid bb)(b\mid c)^a)^\ast (b\mid c\mid a(a\mid ba)^\ast (c\mid bb))^\ast\]

Equivalent regular expression with \{\mid, \cdot, \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)eiΣ*)
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

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

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

\[ L_3 = L_1 \cap L_2 \]

(0,0) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (2,2)
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
Finite state transducers
Recap: anatomy of an FST

Formal definition

\( Q = \{0,1,2,3\} \) (set of states)
\( \Sigma = \{a,b,c,d\} \) (alphabet)
\( q_0 = 0 \) (initial state)
\( F = \{0,1,2\} \) (set of final states)
\( \delta \) (transition function)
Recap: anatomy of an FST

**Graph representation**

![Graph representation of an FST]

**Interpretation**

- An FST defines a set of string pairs (a relation)
  - In this case $T=\{(a,a),(b,b),(c,c), (cad,cdb), \ldots\}$
  - 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 \times L_2 (cross-product)

T \circ U (composition)
Algebraic operations on transducers

T U (concatenation)

T | U (union)

T* (Kleene closure)

rev(T) (reversal)

L₁ x L₂ (cross-product)

T o U (composition)
Algebraic operations on transducers

\[ T \cup U \text{ (concatenation)} \]

\[ T \mid U \text{ (union)} \]

\[ T^* \text{ (Kleene closure)} \]

\[ \text{rev}(T) \text{ (reversal)} \]

\[ L_1 \times L_2 \text{ (cross-product)} \]

\[ T \circ U \text{ (composition)} \]
Composition: product construction

$T_1$

$T_2$

$b:x$ $d:d$

$T_3 = T_1 \circ T_2$

$T_1$

$T_2$

$T_3$

$(0,0)$ $(1,0)$ $(2,0)$
String rewriting operators

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

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

Example: \((a|e|i|o|u) \to 0 / \_ \#\)
deleate vowels at the ends of words

Difficult to implement correctly in the general case
Modeling morphology and phonology

epäjärjestelmällistymättömyydelläänsäkaänköhän

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

Modeled by a step-by-step generative process:

`un`+'system'    +`ize`

epä+järjestelmä+lis+...

UNDERLYING REPRESENTATION

Lexical rules

↓

LEXICAL REPRESENTATION

Postlexical rules

↓

SURFACE REPRESENTATION

put morphemes together

phonemes and morphemes change when they are conjoined, modeled by phonological rules.
“Generative” word model

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

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
remove boundaries

impossibility

(2) Apply sound change rules + orthographic rules
Four tricks to model this

(1) Extended operators (Booleans, replacements)

(2) Use alphabet independent algorithms

\[ \Sigma^* \ a \ \Sigma^* \]

(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

in+possible+ity

change n to m before p (nasal assimilation)

im+possible+ity

ble+ity → bility

im+possibility

remove boundaries

impossibility

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

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

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

(2) Apply sound change rules + orthographic rules

...then compose
Composition

im+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
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

Solution: make lexicon a transduction:

IN: NEG+possible+ity+NOUN+PLURAL

Lex. transducer

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

im+possible+ity+s

im+possible+ity+s

im+possibility+s

impossibilities
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

*See also: https://code.google.com/p/foma/wiki/MorphologicalAnalysisTutorial
Toy grammar of English

Toy lexicon: kiss, hire, spy
Possible suffixes: ed, ing, s
Generate kiss+s/kisses, spy+ed/spied, hire+ing/hiring, hire+ed/hired, etc.
Some derivations

- hire+ing
  - Edelete
  - Einsert
  - Delete +
  - hiring

- hire+ed
  - Edelete
  - Einsert
  - Delete +
  - hired

- kiss+s
  - Edelete
  - Einsert
  - Delete +
  - kisses
# Compile with foma -l analyzer.foma

def Stems spy | kiss | hire;  # Lexicon
def Suffixes "+" [ 0 | s | ed | ing ];  # Suffixes

def Lexicon Stems Suffixes ;

def YRule1 y -> i e | _ "+" s ;  # spy+s > spie+s
def YRule2 y -> i | _ "+" ed ;  # spy+ed > spi+ed
def Einsert "+" -> e | s s ;  # kiss+s > kisses
def Edelete e -> 0 | _ "+" [e|i];  # hire+ed > hir+ed, hire+ing > hir+ing
def Cleanup "+" -> 0 ;  # hir+ing > hiring, etc.

def Grammar Lexicon .o. YRule1 .o. YRule2 .o. Einsert .o. Edelete .o. Cleanup;
regex Grammar;

# Test with e.g. "up spies"
# Compile with foma -l analyzer2.foma

def Stems    s p y | k i s s | h i r e ;
def Suffixes 0: "+" [ "[INF]" : 0 | "[NOUN][SINGULAR]" : 0 | "[PRES]" : s | "[NOUN][PLURAL]" : s | "[PASTPART]" : [e d] | "[PRESPART]" : [i n g] ];

def Lexicon     Stems Suffixes ;

def YRule1       y -> i e || _ "+" s ;   # spy+s > spie+s
def YRule2       y -> i || _ "+" e d ;   # spy+ed > spi+ed
def Einsert     "+" -> e || s _ s ;      # kiss+s > kisses
def Edelete      e -> 0 || _ "+" [e|i] ;  # hire+ed > hir+ed, hire+ing > hir+ing
def Cleanup     "+" -> 0 ;               # hir+ing > hiring, etc.

def Grammar  Lexicon .o. YRule1 .o. YRule2 .o. Einsert .o. Edelete .o. Cleanup;
regex Grammar;

# Test with e.g. "up spies"
The 2 second spell checker

NEG+possible+ity+NOUN+PLURAL

(1) Extract the possible outputs of the “Grammar” transducer, and convert to automaton (output-side projection)

(2) Test a word against automaton
The 5 second spelling corrector

Assume we have a list of words as a repeating FST as before

hired

W

hired
The 5 second spelling corrector

Assume we have a list of words as a repeating FST as before

Now, create a transducer $C_1$ that makes one change in a word (one deletion, one change, one insertion)

$$\text{abc}$$

$$\text{ab, bc, ac, aba, aac, abca, ...}$$
The 5 second spelling corrector

Compose

hired

W

hired

CI

xire, hird, hire, hiredx, ired, hied,...
The 5 second spelling corrector

Compose

W o Cl

xire, hird, hire, hiredx, ired, hied,...
# Simple spelling corrector
# Compile with foma -l analyzer3.foma

def Stems spy | kiss | hire;
def Suffixes 0:"+" [ "[INF]":0 | "[NOUN][SINGULAR]":0 | "[PRES]":s | "[NOUN][PLURAL]":s | "[PASTPART]":[e d] | "[PRESPART]":[i n g] ];
def Lexicon Stems Suffixes;

def YRule1 y -> i e || _ "+" s ; # spy+s > spie+s
def YRule2 y -> i || _ "+" e d ; # spy+ed > spi+ed
def Einsert "+" -> e || s _ s ; # kiss+s > kisses
def Edelete e -> 0 || _ "+" [e|i]; # hire+ed > hir+ed, hire+ing > hir+ing
def Cleanup "+" -> 0 ; # hir+ing > hiring, etc.
def Grammar Lexicon .o. YRule1 .o. YRule2 .o. Einsert .o. Edelete .o. Cleanup;

def C1 ?* [?:0|0:?|?:?:-?] ?* ; # Change one symbol (delete, insert, or change)
regex Grammar.2 .o. C1; # .2 is extraction of output side

# Test with e.g. "up hird"
Can also use a word list for creating a corrector

```python
def Grammar;
def C1 ?* [?:0|0:?|?:?-?] ?* ;
define C1: 354 bytes. 2 states, 5 arcs, Cyclic.

regex Grammar .o. C1;

foma[0]: up
apply up> hird
bird
third
hard
hired
hire
hind
hid
herd
gird
apply up>
```
Entirely non-orthographic grammar

def Stems s p Λι | k ι s | h Λι r ;
def Suffixes 0:”+” [ ”[INF]”:0 | ”[PRES]”:z | ”[PASTPART]”:[d] | ”[PRESPART]”:[ι η] ];

def Sib [s|z];  # Sibilants

def Unvoiced [h|s];  # Unvoiced phonemes

define ObsAssimilation d -> t  ||  Unvoiced ”+” _ ;
define Epenthesis [..] -> ι  ||  Sib ”+” _ Sib ;
define Cleanup ”+” -> 0;

def Lexicon    Stems Suffixes ;

def Grammar    Lexicon .o. ObsAssimilation .o. Epenthesis .o. Cleanup;

regex Grammar;
Wrapup

• The above are standard techniques - morphological/phonological grammars have been written for hundreds of languages in this way.

• The calculus is crucial - thinking about states and transitions is counterproductive.

• A well-designed grammar should be very accurate, barring misspellings (easily >99% recall).

• There are also probabilistic extensions to all of the above (to combine with language models, to handle noisy data, etc.).

• These grammars are also used to improve POS-taggers, parsers, chunkers, named entity recognition, etc.
Class announcement: Machine Learning and Linguistics

LING 3800/6300
Spring 2015

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