CMPT 413
Computational Linguistics

Anoop Sarkar
http://www.cs.sfu.ca/~anoop
Finite-state transducers

• Many applications in computational linguistics

• Popular applications of FSTs are in:
  – Orthography
  – Morphology
  – Phonology

• Other applications include:
  – Grapheme to phoneme
  – Text normalization
  – Transliteration
  – Edit distance
  – Word segmentation
  – Tokenization
  – Parsing
Orthography and Phonology

• Orthography: written form of the language (affected by morpheme combinations)
  move + ed → moved
  swim + ing → swimming S W IH1 M IH0 NG

• Phonology: change in pronunciation due to morpheme combinations (changes may not be confined to morpheme boundary)
  intent IH2 N T EH1 N T + ion
  → intention IH2 N T EH1 N CH AH0 N
Orthography and Phonology

• Phonological alternations are not reflected in the spelling (orthography):
  – Newton Newtonian
  – maniac maniacal
  – electric electricity

• Orthography can introduce changes that do not have any counterpart in phonology:
  – picnic picnicking
  – happy happiest
  – gooey gooiest
Segmentation and Orthography

• To find entries in the lexicon we need to segment any input into morphemes
• Looks like an easy task in some cases:
  \( \text{looking} \rightarrow \text{look} + \text{ing} \)
  \( \text{rethink} \rightarrow \text{re} + \text{think} \)
• However, just matching an affix does not work:
  \*\( \text{thing} \rightarrow \text{th} + \text{ing} \)
  \*\( \text{read} \rightarrow \text{re} + \text{ad} \)
• We need to store valid stems in our lexicon
  what is the stem in \text{assassination} (\text{assassin} and not \text{nation})
Porter Stemmer

• A simpler task compared to segmentation is simply stripping out all affixes (a process called stemming, or finding the stem)
• Stemming is usually done without reference to a lexicon of valid stems
• The Porter stemming algorithm is a simple composition of FSTs, each of which strips out some affix from the input string
  – input=..ational, produces output=..ate (relational → relate)
  – input=..V..ing, produces output=ε (motoring → motor)
Porter Stemmer

- False positives (stemmer gives incorrect stem):
  - doing → doe, policy → police
- False negatives (should provide stem but does not):
  - European → Europe, matrices → matrix

I’m a rageaholic. I can’t live without rageahol.

Homer Simpson, from The Simpsons

- Despite being linguistically unmotivated, the Porter stemmer is used widely due to its simplicity (easy to implement) and speed
Segementation and orthography

- More complex cases involve alterations in spelling
  
  *foxes → fox + s [e-insertion]*
  
  *loved → love + ed [e-deletion]*
  
  *flies → fly + s [i to y, e-deletion]*
  
  *panicked → panic + ed [k-insertion]*
  
  *chugging → chug + ing [consonant doubling]*
  
  *singging → sing + ing*
  
  *impossible → in + possible [n to m]*

- Called *morphographemic* changes.

- Similar to but not identical to changes in pronunciation due to morpheme combinations
Morphological Parsing with FSTs

• Think of the process of decomposing a word into its component morphemes in the reverse direction: as generation of the word from the component morphemes

• Start with an abstract notion of each morpheme being simply combined with the stem using concatenation
  – Each stem is written with its part of speech, e.g. cat+N
  – Concatenate each stem with some suffix information, e.g. cat+N+PL
  – e.g. cat+N+PL goes through an FST to become cats (also works in reverse!)
Morphological Parsing with FSTs

- Retain simple morpheme combinations with the stem by using an intermediate representation:
  - e.g. cat+N+PL becomes \textit{cat}^s#
- Separate rules for the various spelling changes. Each spelling rule is a different FST
- Write down a separate FST for each spelling rule

\begin{align*}
\textit{foxes} & \rightarrow \text{fox}^s# \quad [\textit{e}\text{-insertion FST}] \\
\textit{loved} & \rightarrow \text{love}^\text{ed}# \quad [\textit{e}\text{-deletion FST}] \\
\textit{flies} & \rightarrow \text{fly}^s# \quad [\textit{i} \text{ to } \textit{y}, \textit{e}\text{-deletion FST}] \\
\textit{panicked} & \rightarrow \text{panic}^\text{ed}# \quad [\textit{k}\text{-insertion FST}] \\
\text{etc.}
\end{align*}
Lexicon FST (stores stems)

Compose the above lexicon FST with some inflection FST
This machine relates intermediate forms like $\text{fox}^{\wedge}s\#$ to underlying lexical forms like $\text{fox}+N+\text{PL}$

**Lexical**

```
  f  o  x  +N  +PL
```

**Intermediate**

```
  f  o  x  ^s  #
```
The label *other* means pairs not use anywhere in the transducer.

Since # is used in a transition, $q_0$ has a transition on # to itself.

States $q_0$ and $q_1$ accept default pairs like $(cat^s#, cats#)$

State $q_5$ rejects incorrect pairs like $(fox^s#, foxs#)$
e-insertion FST

• Run the e-insertion FST on the following pairs:
  
  \((\text{fir}^#, \text{fir}^#)\) \hspace{1cm} (\text{fizz}^s#, \text{fizz}s#)
  
  \((\text{fir}^s#, \text{firs}#)\) \hspace{1cm} (\text{fizz}^s#, \text{fizzes#})
  
  \((\text{fir}^s#, \text{fires}#)\) \hspace{1cm} (\text{fizz}^\text{ing}#, \text{fizzing#})

• Find the state the FST reaches after attempting to accept each of the above pairs

• Is the state a final state, i.e. does the FST accept the pair or reject it
• We first use an FST to convert the lexicon containing the stems and affixes into an intermediate representation
• We then apply a spelling rule that converts the intermediate form into the surface form
• **Parsing**: takes the surface form and produces the lexical representation
• **Generation**: takes the lexical form and produces the surface form
• But how do we handle multiple spelling rules?

![Diagram showing Lexical, Intermediate, and Surface forms]

- **Lexical**: fox +N +PL
- **Intermediate**: fox asm #
- **Surface**: foxes
Method 1: Composition

FST composition: creates one FST for all rules

write one FST for each spelling rule: each FST has to provide input to next stage

.. y+s
.. ies

FST
FST_1
FST_2
iable
FST_n
.. ies

Lexicon
Method 2: Intersection

Creating one FST implies we have to do **FST intersection** (but there’s a catch: what is it?)

Write each FST as an equal length mapping ($\varepsilon$ is taken to be a real symbol)
Intersecting/Composing FSTs

- Implement each spelling rule as a separate FST
- We need slightly different FSTs when using Method 1 (composition) vs. using Method 2 (intersection)
  - In Method 1, each FST implements a spelling rule if it matches, and transfers the remaining affixes to the output (composition can then be used)
  - In Method 2, each FST computes an equal length mapping from input to output (intersection can then be used). Finally compose with lexicon FST and input.
- In practice, composition can create large FSTs
Length Preserving “two-level” FST for \textit{e-deletion}

Stems/Lexicon

move + ed
move \(\varepsilon\) \(\varepsilon\) ed

\(\text{other}_1 = \Sigma - \{e,v\}\)
\(\text{other}_2 = \Sigma - \{e,v, +\}\)
Rewrite Rules

• Context dependent rewrite rules: $\alpha \rightarrow \beta / \lambda \_\_ \rho$
  
  - $(\lambda \alpha \rho \rightarrow \lambda \beta \rho; \text{ that is } \alpha \text{ becomes } \beta \text{ in context } \lambda \_\_ \rho)$
  - $\alpha, \beta, \lambda, \rho$ are regular expressions, $\alpha = \text{ input}, \beta = \text{ output}$

• How to apply rewrite rules:
  
  - Consider rewrite rule: $a \rightarrow b / ab \_\_ ba$
  
  - Apply rule on string $abababababa$
  
  - Three different outcomes are possible:
    
    - $abbbabbbababa$ (left to right, iterative)
    - $ababbbbabba$ (right to left, iterative)
    - $abbbbbbabbba$ (simultaneous)
Rewrite Rules

\[ u \rightarrow i / i C^* \_ \]

\[ (u \rightarrow i / \Sigma^* i C^* \_ \Sigma^*) \]

Input: kikukuku

from (R. Sproat slides)
Rewrite Rules

\[ u \rightarrow i / i C^* \_ \_ \]

\[ kikukuku \]
\[ kikukuku \]
\[ kikikikuku \]
\[ kikikikiku \]
\[ kikikiki \]

output of one application feeds next application

\( \rightarrow \text{left to right application} \)
Rewrite Rules

\[ u \rightarrow i / i C^* \ \\
\text{kikukuku}
\text{kikukuku}
\text{kikukuku}
\text{kikikuku}
\text{kikikiki} \]

right to left application
Rewrite Rules

\[ u \rightarrow i / i C^* \_ \_ \text{ kikukuku kikukuku kikikukuku} \]

*simultaneous application*
(context rules apply to input string only)
Rewrite Rules

- Example of the e-insertion rule as a rewrite rule:
  \[ \varepsilon \rightarrow e / (x \mid s \mid z)^\_ s# \]
- Rewrite rules can be optional or obligatory
- Rewrite rules can be ordered \text{wrt} each other
- This ensures exactly one output for a set of rules
Rewrite Rules

• Rule 1: iN → im / __ (p | b | m)
• Rule 2: iN → in / __
• Consider input iNpractical (N is an abstract nasal phoneme)
• Each rule has to be obligatory or we get two outputs: *impractical* and *inpractical*
• The rules have to be ordered wrt to each other so that we get *impractical* rather than *inpractical* as output
• The order also ensures that *intractable* gets produced correctly
Rewrite Rules

• Under some conditions, these rewrite rules are equivalent to FSTs
• We cannot apply output of a rule as input to the rule itself iteratively:
  \[ \varepsilon \rightarrow ab / a \_\_ b \]
  If we allow this, the above rewrite rule will produce \( a^n b^n \) for \( n \geq 1 \) which is not regular
  Why? Because we rewrite the \( \varepsilon \) in \( a\varepsilon b \) which was introduced in the previous rule application
  Matching the \( a\_\_b \) as left/right context in \( a\varepsilon b \) is ok
Rewrite Rules

• In a rewrite rule: $\alpha \rightarrow \beta \lambda \rho$

• Rewrite rules are interpreted so that the input $\alpha$ does not match something introduced in the previous rule application

• However, we are free to match the context either $\lambda$ or $\rho$ or both with something introduced in the previous rule application (see previous examples)

• In this case, we can convert them into FSTs
Rewrite rules to FSTs

\( u \rightarrow i / \Sigma^* i C^* \_ \Sigma^* \)  
(example from R. Sproat’s slides)

- Input: kikukupapu (use left-right iterative matching)
- Mark all possible right contexts
  \( > k > i > k > u > k > u > p > a > p > u > \)
- Mark all possible left contexts
  \( > k > i <> k <> u > k > u > p > a > p > u > \)
- Change \( u \) to \( i \) when delimited by \(<>\)
  \( > k > i <> k <> i > k > u > p > a > p > u > \)
- But the next \( u \) is not delimited by \(<>\) and so cannot be changed even though the rule matches
Rewrite rules to FSTs

\[ u \rightarrow i / \Sigma^* i C^* \_ \_ \Sigma^* \]

- Input: kikukupapu
- Mark all possible right contexts
  \[ > k > i > k > u > k > u > p > a > p > u > \]
- Mark all \( u \) followed by \( > \) with \( <_1 \) and \( <_2 \)
  \[ k > i > k <_1 > u > k <_1 > u > p > a > p <_1 > u > \]
  \[ <_2 u <_2 u <_2 u \]
- Change all \( u \) to \( i \) when delimited by \( <_1 \)
  \[ k > i > k <_1 > i > k <_1 > i > p > a > p <_1 > i > \]
  \[ <_2 u <_2 u <_2 u \]
u → i / Σ* i C* ___ Σ*

Rewrite rules to FSTs

\[
\begin{align*}
&k > i > k <_1 > i > k <_1 > i > p > a > p <_1 > i > \\
&<_2 u <_2 u <_2 u
\end{align*}
\]

- Delete >
  \[
  \begin{align*}
  &k i k <_1 i k <_1 i p a p <_1 i \\
  &<_2 u <_2 u <_2 u
  \end{align*}
  \]

- Only allow \( i \) where \( <_1 \) is preceded by \( iC* \), delete \( <_1 \)
  \[
  \begin{align*}
  &k i k i k i p a p \\
  &<_2 u <_2 u <_2 u
  \end{align*}
  \]

- Allow only strings where \( <_2 \) is not preceded by \( iC* \), delete \( <_2 \)
  \[
  \begin{align*}
  &k i k i k i p a p u
  \end{align*}
  \]
Rewrite rules to FST

• For every rewrite rule: $\alpha \rightarrow \beta / \lambda _\_ \rho$:
  - FST $r$ that inserts $>$ before every $\rho$
  - FST $f$ that inserts $<_1$ & $<_2$ before every $\alpha$ followed by $>$
  - FST $replace$ that replaces $\alpha$ with $\beta$ between $<_1$ and $>$ and deletes $>$
  - FST $\lambda _1$ that only allows all $<_1 \beta$ preceded by $\lambda$ and deletes $<_1$
  - FST $\lambda _2$ that only allows all $<_2 \beta$ not preceded by $\lambda$ and deletes $<_2$

• Final FST = $r \circ f \circ replace \circ \lambda _1 \circ \lambda _2$

• This is only for left-right iterative obligatory rewrite rules: similar construction for other types
Rewrite Rules to FST

FST for replace

Create a new FST by taking the cross product of the languages $\alpha$ and $\beta$ and each state of this new FST: $[\alpha \times \beta]$ has loops for the transitions $<_1: \varepsilon, <_2: \varepsilon, >: \varepsilon$
Ambiguity (in parsing)

- Global ambiguity: (de+light+ed vs. delight+ed)
  
  foxes → fox+N+PL (*I saw two foxes*)
  
  foxes → foxes+V+3SG (*Clouseau foxes them again*)

- Local ambiguity:

  *assess* has a prefix string *asses* that has a valid analysis:
  
  asses → ass+N+PL

- Global ambiguity results in two valid answers, but local ambiguity returns only one.

- However, local ambiguity can also slow things down since two analyses are considered partway through the string.
Summary

- FSTs can be applied to creating lexicons that are aware of morphology
- FSTs can be used for simple stemming
- FSTs can also be used for morphographemic changes in words (spelling rules), e.g. fox+N+PL becomes foxes
- Multiple FSTs can be composed to give a single FST (that can cover all spelling rules)
- Multiple FSTs that are length preserving can also be run in parallel with the intersection of the FSTs
- Rewrite rules are a convenient notation that can be converted into FSTs automatically
- Ambiguity can exists in the lexicon: both global & local
\[ [C]' = [C]-\{n\} \]

other = \( \Sigma-[C]'-\{n,e\} \)