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:
  - looking → look + ing
  - rethink → re + think
- However, just matching an affix does not work:
  - *thing → th + ing
  - *read → re + ad
- We need to store valid stems in our lexicon
  - what is the stem in assassination (assassin and not 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):
  \(doing \rightarrow doe, policy \rightarrow police\)

• False negatives (should provide stem but does not):
  \(European \rightarrow Europe, matrices \rightarrow matrix\)

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

Segmentation and orthography

• More complex cases involve alterations in spelling
  \(foxes \rightarrow fox + s \quad [e\text{-insertion}]\)
  \(loved \rightarrow love + ed \quad [e\text{-deletion}]\)
  \(flies \rightarrow fly + s \quad [y\ to\ i, e\text{-insertion}]\)
  \(panicked \rightarrow panic + ed \quad [k\text{-insertion}]\)
  \(chugging \rightarrow chug + ing \quad [\text{consonant doubling}]\)
  \(*\text{signging} \rightarrow \text{sing} + \text{ing}\)
  \(impossible \rightarrow \text{in} + \text{possible} \quad [n\ to\ m]\)

• 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 *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

  *foxes* :: fox^s# [ *e-insertion* FST ]

  *loved* :: love^ed# [ *e-deletion* FST ]

  *flies* :: fly^s# [ *y to i, e-insertion* FST ]

  *panicked* :: panic^ed# [ *k-insertion* FST ] (arced::arc^ed#)??

  *etc.*
Lexicon FST (stores stems)

Compose the above lexicon FST with some inflection FST

This machine relates intermediate forms like fox^s# to underlying lexical forms like fox+N+PL

**Lexical**

| 2 | f | o | x | +N | +PL |

**Intermediate**

| 2 | 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:
  
  $(fir#, fir#)$  
  $(fiz^s#, fizzes#)$  
  $(fiz^s#, firs#)$  
  $(fiz^s#, fires#)$  
  $(fiz^s#, fires#)$  
  $(fiz^s#, fizzes#)$  
  $(fiz^s#, fizzes#)$  
  $(fiz^s#, fizzes#)$  
  $(fiz^s#, fizzes#)$

• 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?

Method 1: Composition

**Lexicon**

**Intermediate**

**Surface**

---

---
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 (ε 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 *e-deletion*

**Stems/Lexicon**

```
move  ^ ed
move  ε ed
other₁ = Σ - {e,v}
other₂ = Σ - {e,v,^}
```

Should also work for leaving :: leave^ing

---

**Motivation for using FSTs**

- We have provided a formal device of FSTs that enables “finite-state” translations
- Translations of this kind are useful in many different contexts in computational linguistics (and beyond)
- But why use such a theoretically well-defined model -- why not use common programming language devices for translation?
REGEX v.s. FST

- The common method for string translations is the REGEX extension of regular expressions: allows match & replace
- For example, to perform e-insertion we would:
  ```python
  > infstem = 'fox+N+PL'
  > inter = re.sub('\+N\+PL$', '^s', infstem)
  > inter == 'fox\s'
  > final = re.sub('([sxz])\^s\#', r'\1es', inter)
  > final == 'foxes'
  ```
- Seems simple enough -- why bother with FSTs?
- REGEX algorithms are exponential-time, FSTs are linear time -- sometimes theory is useful in practice!
- Can we retain the useful notation of REGEX expressions?

---

Rewrite Rules

- Context dependent rewrite rules: \( \alpha \rightarrow \beta / \lambda \_\_ \rho \)
  - \( \lambda, \alpha, \beta, \rho \) are regular expressions, \( \alpha = \text{input}, \beta = \text{output} \)
  - e.g. \( \alpha = (alb) \) means input is either \( a \) or \( b \), and \( \beta = (alb) \) means the output is ambiguous: should be either \( a \) or \( b \)
- How to apply rewrite rules:
  - Consider rewrite rule: \( a \rightarrow b / ab \_\_ ba \)
  - Apply rule on string \( abababababa \)
  - Three different outcomes are possible:
    - \( abbbabbbaba \) (left to right, iterative)
    - \( ababbbababa \) (right to left, iterative)
    - \( abbbabbbaba \) (simultaneous)
Rewrite Rules

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

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

Input: kikukuku

---

Rewrite Rules

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

\[ \text{kikukuku} \quad \text{kikukuku} \quad \text{kikikuku} \quad \text{kikikuku} \quad \text{kikikiku} \quad \text{kikikiku} \quad \text{kikikiki} \]

output of one application feeds next application

\[ \text{left to right application} \]
Rewrite Rules

u → i / i C* __  kikukuku
    kikukuku
    kikukuku
    kikikiku
    kikikiku
    kikikiki

right to left application

Rewrite Rules

u → i / i C* __  kikukuku
    kikukuku
    kikukuku
    kikikiku
    kikikiku

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 \textit{wrt} each other
• This ensures exactly one output for a set of rules

Rewrite Rules

• Rule 1: iN \rightarrow im / \_\_ (p \mid b \mid m)
• Rule 2: iN \rightarrow in / \_\_ 
• Consider input \textit{iNpractical} (N is an abstract nasal phoneme)
• Each rule has to be obligatory or we get two outputs: \textit{impractical} and \textit{inpractical}
• The rules have to be ordered \textit{wrt} each other so that we get \textit{impractical} rather than \textit{inpractical} as output
• The order also ensures that \textit{intractable} gets produced correctly
Example: Finnish Harmony

<table>
<thead>
<tr>
<th>Gloss</th>
<th>Nominative</th>
<th>Partitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>sky</td>
<td>taivas</td>
<td>taivas+ta</td>
</tr>
<tr>
<td>telephone</td>
<td>puhelin</td>
<td>puhelin+ta</td>
</tr>
<tr>
<td>plain</td>
<td>lakeus</td>
<td>lakeut+ta</td>
</tr>
<tr>
<td>reason</td>
<td>syy</td>
<td>syy+tä</td>
</tr>
<tr>
<td>short</td>
<td>lyhyt</td>
<td>lyhyt+tä</td>
</tr>
<tr>
<td>friendly</td>
<td>ystävällinen</td>
<td>ystävällinen+tä</td>
</tr>
</tbody>
</table>

*i.e are neutral wrt harmony*

- talossansakaanko ‘not in his house either?’
- kynässäsäkäänkö ‘not in his pen either?’

Rewrite Rules

- Context dependent rewrite rules: \( \alpha \rightarrow \beta / \lambda \ _\rho \)
- Can express context sensitive rules or regular relations
- Computational constraints on rewrite rules:
  - Consider rewrite rule: \( c \rightarrow acb / a \ _\ b \)
  - Apply left to right iteratively on base-form \( c \)
  - Produces a sequence of strings:

\[
\begin{array}{cccccc}
a & a & a & c & b & b & b \\
\end{array}
\]

*Do we need such long-distance effects in morphophonological rules?*
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).

• Impose a simple constraint on how rewrite rules are applied: output cannot be re-written.
  
  e.g. $c \rightarrow a \_ b / a \_ b$

Rewrite Rules

• We cannot apply output of a rule as input to the rule itself iteratively:
  
  $c \rightarrow a \_ b$
  
  If we allow this, the above rewrite rule will produce $a^n c b^n$ for $n \geq 1$ which is not regular.

  Why? Because we rewrite the $c$ in $acb$ which was introduced in the previous rule application.

  Matching the $\_ \_ b$ as left/right context in $acb$ is ok.

• Kaplan and Kay constraints:
  
  – Constraint ensures rewrite rules are equivalent to regular relations.
  
  – Naturally expresses the local nature of “finite-state” translation.

  – Under these conditions, these rewrite rules are equivalent to FSTs.
Rewrite rules to FSTs

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

*...kikukuku
√kikikikiki

Rewrite Rules to FSTs

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

*...kikukuku
√kikikikiki
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 \( <_1 > \)
  \[ k >i >k ><_1 i >k ><_1 i >p >a >p ><_1 i > <_2 u <_2 u <_2 u \]
- Delete \( > \)
  \[ k i k <_1 i k <_1 i p a p <_1 i <_2 u <_2 u <_2 u \]
- Only allow \( i \) where \( <_1 \) is preceded by \( iC^* \), delete \( <_1 \)
  \[ k i k i k i p a p <_2 u <_2 u <_2 u \]
- Allow only strings where \( <_2 \) is \textbf{not} preceded by \( iC^* \), delete \( <_2 \)
  \[ k i k i k i p a p u \]
Rewrite Rules to FST

- Mark right contexts: $a > b$ $a > b > b$
- Mark $a$ and $b$ before $>$ with $<_1$ and $<_2$
  $$<_1 a > b <_1 a > b > b$$
  $$<_2 a < _<_2 a < _<_2 b$$
- Match $<_1 \text{LHS} >$ and convert to $<_1 \text{RHS} >$; delete $>$
  $$<_1 b b < _<_1 b < _<_1 a b$$
  $$<_2 a < _<_2 a < _<_2 b$$
- Allow $<_1 \text{RHS}$ when left context exists; delete $<_1$
  $$<_1 b b < _<_1 b < _<_1 a b = _<_2 a b (b \parallel _<_2 a) (a \parallel _<_2 b) b$$
  $$<_2 a < _<_2 a < _<_2 b$$
- Allow $<_2 \text{LHS}$ when left context does not exist; delete $<_2$
  $$a b b a b$$

Rewrite rules to FST

- For every rewrite rule: $\alpha \rightarrow \beta \lambda \rho$
  - FST $r$ that inserts $>$ before every $\rho$
    $$r = \varepsilon \rightarrow > / \Sigma^* \rho$$
  - FST $f$ that inserts $<_1 \& <_2$ before every $\alpha$ followed by $>$
    $$f = \varepsilon \rightarrow (\{<_1\} \cup \{<_2\}) / (\Sigma \cup \{>\})^* \alpha >$$
    where $\alpha >$ freely allows $>$ anywhere in $\alpha$
  - FST $\text{replace}$ that replaces $\alpha$ with $\beta$ between $<_1$
    and $>$ and deletes $>$
    for $\text{replace}$ we write a special cross product FST
Rewrite Rules to FST

FST for replace

Create a new FST by taking the cross product of the languages $\alpha$ and $\beta$ (every string in $\alpha$ is mapped to every string in $\beta$).

Note that while matching $\alpha$ we need to ignore all the instances of $>$, $<$, $\leq$, $\geq$ we previously inserted.

Rewrite rules to FST

- FST $\lambda_1$ that only allows all $<_1 \beta$ preceded by $\lambda$ and deletes $<_1$
  \[
  \lambda_1 = <_1 \rightarrow \epsilon / \#\Sigma^*\lambda \_ \epsilon
  \]
  where $#$ is a symbol marking start of the string and we ignore the $<_2$ symbols in the string
- FST $\lambda_2$ that only allows all $<_2 \beta$ not preceded by $\lambda$ and deletes $<_2$
  \[
  \lambda_2 = <_2 \rightarrow \epsilon / \#\text{complement}(\Sigma^*\lambda) \_ \epsilon
  \]
- Final FST $= rof o replace o \lambda_1 o \lambda_2$
- This is only for left-right iterative obligatory rewrite rules: similar construction for other types
Ambiguity (in parsing)

- Global ambiguity: (de+light+ed vs. delight+ed)
  
  \[ foxes \rightarrow \text{fox+N+PL} \ (I \text{ saw two foxes}) \]
  
  \[ foxes \rightarrow \text{foxes+V+3SG} \ (Clouseau foxes them again) \]

- Local ambiguity:
  
  \[ assess \text{ has a prefix string asses that has a valid analysis: } \]
  
  \[ asses \rightarrow \text{ass+N+PL} \]

- 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 exist in the lexicon: both global & local
\[ \text{other} = \Sigma - [C]' - \{n,e\} \]

\[ [C]' = [C] - \{n\} \]