A tutorial on finite-state text processing

Kyle Gorman
City University of New York
Outline

• Formal preliminaries
• OpenFst and friends:
  • the past…
  • …and the future…
• Key FST algorithms
• Two worked examples
Formal preliminaries
(image: credit: Wikimedia Commons)
A state diagram with three states labeled 0, 1, and 2. The transitions are:
- From 0 to 1 with label "turn-knob:ε"
- From 0 to 2 with label "ε:emit-gumball"
- From 1 to 2 with label "turn-knob:ε"
- Transition from 1 to 1 with label "insert-coin:ε"
A set is an abstract, unordered collection of distinct objects, the *members* of that set. By convention capital italic letters denote sets and lowercase letters to denote their members. Set membership is indicated with the $\in$ symbol; e.g., $x \in X$ is read “$x$ is a member of $X$”. The empty set is denoted by $\emptyset$. 
A set $X$ is said to be a *subset* of another set $Y$ just in the case that every member of $X$ is also a member of $Y$. The subset relationship is indicated with the $\subseteq$ symbol; e.g., $X \subseteq Y$ is read as “$X$ is a subset of $Y$”.

http://pynini.opengrm.org
Union and intersection

- The *union* of two sets, $X \cup Y$, is the set that contains just those elements which are members of $X$, $Y$, or both.

$$X \cup Y = \{x : x \in X \lor x \in Y\}$$

- The *intersection* of two sets, $X \cap Y$, is the set that contains just those elements which are members of both $X$ and $Y$.

$$X \cap Y = \{x : x \in X \land x \in Y\}$$
Strings

Let $\Sigma$ be an *alphabet* (i.e., a finite set of symbols). A *string* (or *word*) is any finite ordered sequence of symbols such that each symbol is a member of $\Sigma$. By convention typewriter text is used to denote strings. The empty string is denoted by $\epsilon$ (*epsilon*). String sets are also known as *languages*. 

http://pynini.opengrm.org
Concatenation and closure

- The *concatenation* of two languages, $X Y$, consists of all strings formed by concatenating a string in $X$ with a string in $Y$.

$$X Y = \{xy : x \in X, y \in Y\}$$

- The *closure* of a language, $X^*$, is an infinite language consisting of zero or more “self-concatenations” of $X$ with itself.

$$X^* = \{\epsilon\} \cup X^1 \cup X^2 \cup X^3 \ldots$$

$$= \{\epsilon\} \cup X \cup XX \cup XXX \ldots$$
Regular languages (Kleene, 1956)

- The empty language $\emptyset$ is a regular language.
- The empty string language $\{\varepsilon\}$ is a regular language.
- If $s \in \Sigma$, then the singleton language $\{s\}$ is a regular language.
- If $X$ is a regular language, then its closure $X^*$ is a regular language.
- If $X, Y$ are regular languages, then:
  - their concatenation $XY$ is a regular language, and
  - their union $X \cup Y$ is a regular language.
- Other languages are not regular languages.
Regular languages in the 20th century

Regular languages were first defined by Kleene (1956) and popularized in part by their discussion in the context of the Chomsky(-Schützenberger) hierarchy (e.g. Chomsky and Miller, 1963). Not long afterwards this was followed by two seemingly negative results:

• Phrase structure grammars of the sort used in syntactic theory appear to belong to a higher-classes of formal languages, the mildly context-sensitive languages (Vijay-Shanker et al., 1987).

• The class of regular languages are not “learnable” under Gold’s (1967) notion of language identification in the limit.

http://pynini.opengrm.org
Regular languages in the 21st century

However, an enormous amount of linguistically-interesting phenomena can be described in terms of regular languages (and regular relations)... And, many of these phenomena fall into provably learnable subsets of the regular languages (e.g. Heinz, 2010; Rogers et al., 2010; Chandlee et al., 2014; Jardine and Heinz, 2016; Chandlee et al., 2018).
Finite-state acceptors

An finite-state acceptor (FSA) is a 5-tuple consisting of:

• a set of states $Q$,
• a initial (or “start”) state $s \in Q$,
• a set of final states $F \subseteq Q$,
• an alphabet $\Sigma$, and
• a (partial) transition relation $\delta$ mapping $Q \times (\Sigma \cup \{\epsilon\})$ onto $Q$. 

http://pynini.opengrm.org
Acceptance

If $q' \in \delta(q, \sigma)$, then there exists a transition from $q$ to $q'$ labeled $\sigma$. We can extend $\delta$ using the following recurrence:

$$\forall q \in Q, \forall x \in \Sigma^*, \forall a \in \Sigma \cup \{\epsilon\} : \delta(q, xa) = \delta(\delta(q, x), a)$$

Then, a string $x \in \Sigma^*$ is accepted by the FSA just in the case that $\delta(s, x) \in F$. 

http://pynini.opengrm.org
The cow language /moo+/:

- $Q = \{0, 1, 2, 3\}$
- $s = 0$
- $F = \{3\}$
- $\Sigma = \{m, o\}$
- $\delta = (0, m) \rightarrow \{1\}, (1, 0) \rightarrow \{2\}, (2, 0) \rightarrow \{3\}, (3, 0) \rightarrow 3$
Regular relations

In many cases we are not interested in sets of strings so much as relations or functions between sets of strings. The cross-product of two languages, $X \times Y$ is one such relation: it maps any string in $X$ onto any string in $Y$.

$$X \times Y = \{x \mapsto y : x \in X, y \in Y\}$$

Subsets of the cross-product of two regular languages are known as regular relations.
Finite-state transducers

A finite-state transducer (FST) is a 6-tuple consisting of:

- a set of states $Q$,
- a initial (or “start”) state $s \in Q$,
- a set of final states $F \subseteq Q$,
- an input alphabet $\Sigma$,
- an output alphabet $\Phi$,
- a transition relation $\delta$ mapping $Q \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\})$ onto $Q$. 

http://pynini.opengrm.org
Transduction

If \( q' \in \delta(q, \sigma, \Phi) \) then there exists a transition from \( q \) to \( q' \) with the input label \( \sigma \) and the output label \( \Phi \). We can extend \( \delta \) using the following recurrence:

\[
\forall q \in Q, \forall x \in \Sigma^*, \forall a \in \Sigma, \forall y \in \Phi^*, \forall b \in \Phi : \delta(q, xa, yb) = \delta(\delta(q, x, y), a, b)
\]

Then, an input string \( x \in \Sigma^* \) is *transduced* to an output string \( y \in \Phi^* \)—or, \( x \mapsto y \)—just in the case that \( \delta(s, x, y) \in F \).
{foo} × {bar, baz}
All about $\epsilon$

The $\epsilon$ symbol is a special one which does not match/consume any other symbol. For every FSA we compute an equivalent $\epsilon$-free (or “e-free”) FSA, found using the epsilon-removal algorithm; however, not all FSTs have an equivalent $\epsilon$-free FST.
Weights

We can also add weights to transitions (and final states) subject so long as the weights and their operations define a *semiring* (Mohri, 2002).
A *monoid* is a pair \((\mathbb{K}, \cdot)\) where \(\mathbb{K}\) and \(\cdot\) is an binary operator over \(\mathbb{K}\) such that:

- **closure**: \(\forall a, b \in \mathbb{K} : a \cdot b \in \mathbb{K}\),
- **identity**: \(\exists e \in \mathbb{K}, \forall a \in \mathbb{K} : e \cdot a = a \cdot e = a\), and
- **associativity**: \(\forall a, b, c \in \mathbb{K} : (a \cdot b) \cdot c = a \cdot (b \cdot c)\).

Furthermore, a semiring is said to be *commutative* if
\(\forall a, b \in \mathbb{K} : a \cdot b = b \cdot a\).
A semiring is a five-tuple \((\mathbb{K}, \oplus, \otimes, \bar{0}, \bar{1})\) such that:

- \((\mathbb{K}, \oplus)\) is a commutative monoid with identity element \(\bar{0}\),
- \((\mathbb{K}, \otimes)\) is a monoid with identity element \(\bar{1}\),
- \(\forall a, b, c \in \mathbb{K} : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)\), and
- \(\forall a \in \mathbb{K} : a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}\).
## Common semirings

<table>
<thead>
<tr>
<th></th>
<th>( \mathbb{K} )</th>
<th>( \oplus )</th>
<th>( \otimes )</th>
<th>( \mathring{0} )</th>
<th>( \mathring{1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>( {0, 1} )</td>
<td>( \lor )</td>
<td>( \land )</td>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Probability</td>
<td>( \mathbb{R}_+ )</td>
<td>( + )</td>
<td>( \times )</td>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>Log</td>
<td>( \mathbb{R} \cup {-\infty, +\infty} )</td>
<td>( \oplus_{\log} )</td>
<td>( + )</td>
<td>( +\infty )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>Tropical</td>
<td>( \mathbb{R} \cup {-\infty, +\infty} )</td>
<td>( \min )</td>
<td>( + )</td>
<td>( +\infty )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

NB: \( a \oplus_{\log} b = -\ln(e^{-a} + e^{-b}) \).
Weighted finite-state transducers

We attach weights to transducers in two places:

- We generalize the transition relation $\delta$ so that it maps from $Q \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\})$ onto $Q \times \mathbb{K}$; that is, so that it outputs a weight in addition to a destination state.

- Instead of a set of final states $F$, we add a relation $\omega : Q \rightarrow \mathbb{K}$ giving the final weight for each state.

A state $q \in Q$ is then said to be non-final just in the case that $\omega(q) = \bar{0}$. 

http://pynini.opengrm.org
Weighted transduction

If \((q', k) \in \delta(q, \sigma, \phi)\) then there exists a transition from \(q\) to \(q'\) with the input label \(\sigma\), the output label \(\phi\), and the weight \(k\). In addition to the expected transducer recurrence relation for \(\delta\) defined above, we compute the weight of a path by taking the \(\otimes\)—the product, as defined by the semiring—of the transition weights and the final weight. Thus, given an input string \(x \in \Sigma^*\) is transduced, an output string \(y \in \Phi^*\), when \((q, k) = \delta(s, x, y)\), the weight of the mapping \(x \mapsto y\) is given by \(k \otimes \omega(q)\).
OpenFst and friends
Friends

• The Xerox toolkit (XFST; Beesley and Karttunen 2003)
• The AT&T toolkit (FSM; Mohri et al. 2000)
• Carmel (Knight and Graehl, 1998)
• HFST (Lindén et al., 2013)
• Foma (Hulden, 2009)
• Kleene (Beesley, 2012)
OpenFst (Allauzen et al., 2007)

OpenFst is an open-source C++11 library for weighted finite state transducers developed at Google. Among other things, it is used in:

• Speech recognizers (e.g., Kaldi and many commercial products)
• Speech synthesizers (as part of the “front-end”)
• Input method engines (e.g., mobile text entry systems)

http://pynini.opengrm.org
OpenFst design

There are (at least) four layers to OpenFst:

• A C++ template/header library in <fst/*.h>
• A C++ “scripting” library in <fst/script/*.h,cc>
• CLI programs in /usr/local/bin/*
• A Python extension module pywrapfst

Plus Pynini, an additional extension module that we’ll use below.
OpenGrm

- Baum-Welch (Gorman, forthcoming): CLI tools and libraries for performing expectation maximization on WFSTs
- NGram (Roark et al., 2012): CLI tools and libraries for building conventional n-gram language models encoded as WFSTs
- Thrax (Roark et al., 2012): DSL-based compiler for WFST-based grammar development
- SFst (Allauzen and Riley, 2018): CLI tools and libraries for building *stochastic FSTs*

All these are available under an Apache 2.0 license, and all use the same binary serialization as OpenFst.
Source installation

OpenFst and OpenGrm sources are available online and are regularly tested on Linux (x86_64) and Mac OS X. Windows users should use the Windows Subsytem for Linux (WSL).
Conda installation

Anaconda users can now install OpenFst and OpenGrm (in seconds) using the following command:

```
$ conda install -c conda-forge openfst
```

Also supported are baumwelch, ngram, pynini, and thrax.
OpenFst conventions

- FST and symbol table objects implement copy-on-write (COW) semantics; copy methods and constructors make shallow copies and run in constant-time.
- Iterators are invalidated by mutation.
- Both acceptors and transducers, weighted or unweighted, are represented as weighted transducers.
- $Q$ is a dense integer range starting at zero.
- At most one state can be designated as a start state; an empty FST—one with no states—has a start state of -1.
- Arc labels are non-negative integers; 0 is reserved for $\varepsilon$ and negative integers are reserved for implementation.
- Every state is associated with a final weight; non-final states have an infinite final weight $\bar{0}$ and final states have a non-$\bar{0}$ weight.
Pynini conventions

Some algorithms are inherently *constructive*; others are naturally *destructive*. Pynini adopts the following conventions:

- Constructive algorithms are implemented as module-level functions which return a new FST.
- Destructive algorithms are implemented as instance methods which mutate the instance they’re invoked on. Furthermore:
  - where possible, destructive methods return `self` so that they can be chained, and
  - destructive algorithms also can be invoked constructively using module-level functions.
WFST algorithms
Concatenation

The *concatenation* $AB$ can be computed destructively (on $A$) using $A.concat(B)$ or constructively using $A + B$. The algorithm works by adding an $\epsilon$-arc from every final state in $A$ to the initial state of $B$. 

http://pynini.opengrm.org
aa b
Closure

The closure $A^*$ can be computed destructively using $A.closure()$ or constructively using $\text{closure}(A)$. The algorithm introduces $\epsilon$-arcs from all final states to the initial state.
Union

The union $A \mid B$ can be computed destructively (on $A$) using $A\.union(B)$ or constructively using $A \mid B$. The algorithm introduces an $\epsilon$-arc from the initial state of $A$ to the initial state of $B$. 
dd | eee
The cross-product function transducer constructively computes the cross-product transducer $T = A \times B$. It is defined roughly as follows:

```python
def _transducer(ifst1: Fst, ifst2: Fst) -> Fst:
    upper = arcmap(ifst1, map_type="output_epsilon")
    lower = arcmap(ifst2, map_type="input_epsilon")
    return compose(upper.rmepsilon(),
                   lower.rmepsilon(),
                   compose_filter="match")
```
dd \times eee
Composition

The *composition* \( A \circ B \) can be computed constructively using \( A \odot B \) or \( \text{compose}(A, B) \). By default, non-(co)accessible states are trimmed.
The semantics of composition

- If both $A$ and $B$ are FSAs, $A \circ B$ contains the intersection of all the strings they match.
- If $A$ is an FSA and $B$ is an FST, $A \circ B$ consists of the $B$ relation with its domain restricted to $A$.
- If $A$ is an FST and $B$ is an FSA, $A \circ B$ consists of the $A$ relation with its range restricted to $B$.
- If both $A$ and $B$ are FSTs, $A \circ B$ it produces a chained relation that is equivalent to applying $B$ to the output of $A$. 

http://pynini.opengrm.org
The input projection $\pi_i(A)$, consisting of the domain of transducer $A$, can be computed destructively using $A\cdot\text{project}(\text{False})$ or constructively using $\text{project}(A, \text{False})$.

Similarly, the output projection $\pi_o(A)$, consisting of the range of transducer $A$, can be computed destructively using $A\cdot\text{project}(\text{True})$ or constructively using $\text{project}(A, \text{True})$. 

http://pynini.opengrm.org
The context-dependent rewrite rule compilation function \texttt{cdrewrite} constructively expands an SPE-like phonological rule specification\[
\phi \rightarrow \psi / \lambda \ldots \rho
\]into a transducer using the Mohri and Sproat (1996) algorithm. The algorithm requires us to provide a finite alphabet transducer \((\Sigma \cup \Phi)^*\) over which the rule transducer will operate.
Rule compilation example

The following implements the rule $a \rightarrow b / c \_ c$ over the alphabet \{a, b, c\}.

\[
\begin{align*}
\text{sigma} &= \text{union}("a", "b", "c").\text{closure}() \\
\text{tau} &= \text{transducer}("a", "b") \\
\text{rule} &= \text{cdrewrite}(\text{tau}, "c", "c", \text{sigma})
\end{align*}
\]
Rule application

While not a built-in operation, rule application is traditionally performed as follows:

- Construct an FSA $A$ containing the (union of) input string(s).
- Construct a FST $B$ representing the rule.
- Compute $\pi_o(A \circ B)$ and extract the path(s).

```python
lattice = compose("caccac", rule).project(True)
print(lattice.string())
```
Optimization

An WFST is said to be optimal if it is *minimal*. Minimization algorithms, in turn, require that their input also be *deterministic* (and they preserve that property). In Pynini, Fst objects have a built-in method `optimize` which applies a generic routine for optimization.

[http://pynini.opengrm.org](http://pynini.opengrm.org)
Optimization for unweighted acceptors

The following will produce an optimal FSA for any acyclic acceptor over an idempotent semiring.

```python
def _optimize(fst: Fst) -> Fst:
    opt_props = NO_EPSILONS | I_DETERMINISTIC
    props = fst.properties(opt_props, True)
    fst = fst.copy()
    if not props | NO_EPSILONS:
        fst.rmepsilon()
    if not props | I_DETERMINISTIC:
        fst = determinize(fst)
    return fst.minimize()
```
Advanced optimization

However, **some weighted cyclic FSAs are not determinizable** (Mohri, 1997, 2009). Therefore we determinize and minimize the FSA as if it were an unweighted acceptor. Similarly, **not all transducers are determinizable**. We instead determinize and minimize the WFST as if it were an unweighted acceptor. In both cases, we also perform **arc-sum mapping** as a post-process.
Shortest path

The *shortest path* function `shortestpath` constructively computes the (n-)shortest paths in a WFST. In case of ties, library behavior is deterministic but implementation-defined. The *k-unique paths* can be obtained by determinizing the WFST on the fly. This operation is only well-defined for semirings with the *path property*:
\[ \forall a, b \in \mathbb{K} : a \oplus b \in \{a, b\}. \]
Examples
Rule-based g2p

https://gist.github.com/kylebgorman/124909662f1abdad9a97ef06237c
Speech grammars at Google

Pynini is used extensively at Google for speech-oriented FST grammar development, e.g.:

- Gorman and Sproat (2016) propose an algorithm—implemented in Pynini—which can induce number name grammars from a few-hundred labeled examples.
- Ritchie et al. (2019) describe how Pynini is used to build “unified” verbalization grammars that can be shared by both ASR and TTS.
- Ng et al. (2017) constrain a linear-model-based verbalizers with FST covering grammars.
- Zhang et al. (2019) constrain RNN-based verbalizers with FST covering grammars.

http://pynini.opengrm.org
Some recommended reading

- Sets and strings: Partee et al. 1993, ch. 1–3
- WFST algorithms: Mohri 1997, 2009
- Shortest distance and path problems: Mohri 2002
- Optimizing composition: Allauzen et al. 2010

http://pynini.opengrm.org
More information

http://pynini.opengrm.org


References II


http://pynini.opengrm.org
References III


References IV


References V


http://pynini.opengrm.org
References VI

