A tutorial on finite-state text processing

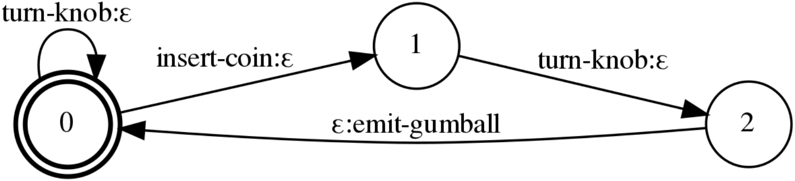
Kyle Gorman
City University of New York
Google Inc.

Outline

- Formal preliminaries
- OpenFst and friends:
    - the past…
    - …and the future…
- Key FST algorithms
- A few worked examples

Formal preliminaries

(image: credit: Wikimedia Commons)


turn-knob:ε
insert-coin:ε
1
turn-knob:ε
0
ε:emit-gumball
2

Sets

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 $\varnothing$.

Subsets

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”.

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.

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

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

$$X \cap Y := \{x : x \in X \wedge x \in Y\}$$

Strings

Let Σ 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 Σ. By convention typewriter text is used to denote strings. The empty string is denoted by ϵ. String sets are also known as *languages*.

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.

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

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

$$\begin{aligned} X^* &:= \{\epsilon\} \cup X^1 \cup X^2 \cup X^3 \ldots \\ &:= \{\epsilon\} \cup X \cup X\,X \cup X\,X\,X \ldots \end{aligned}$$

Regular languages (Kleene, 1956)

- The empty language $\varnothing$ is a regular language.
- The empty string language $\{\epsilon\}$ 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 $X\ Y$ 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:

- Traditional phrase structure grammars belong to a higher class in the hierarchy, the *context-free languages*
- The class of regular languages are not "learnable" under Gold's (1967) notion of *language identification in the limit*.

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 (after Mohri, 1997)

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

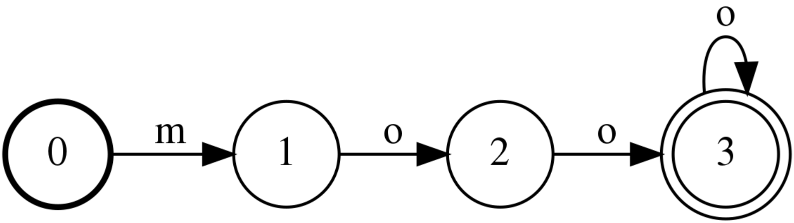
- a set of states Q,
- a initial (or "start") state $s \in Q$,
- a set of final states $F \subseteq Q$,
- an alphabet Σ, and
- a transition relation δ mapping $Q \times (\Sigma \cup \{\epsilon\})$ onto Q.

Acceptance

Let us extend δ using the following recurrence:

$$\forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \delta(q, wa) = \delta(\delta(q, w), a)$$

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


o
m
o
o
0
1
2
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 7-tuple consisting of:

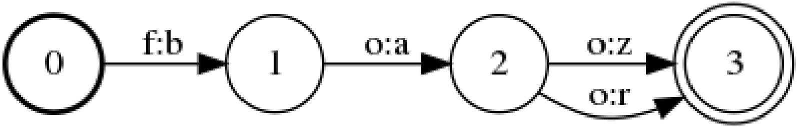
- a set of states Q,
- a initial (or "start") state $s \in Q$,
- a set of final states $F \subseteq Q$,
- an input alphabet Σ,
- an output alphabet Δ,
- a transition relation δ mapping $Q \times (\Sigma \cup \{\epsilon\})$ onto Q.
- an output relation σ mapping $Q \times (\Sigma \cup \{\epsilon\})$ onto Δ^*.

Transduction

We can similarly extend σ using the following recurrence:

$$\forall q \in Q, \forall w \in \Sigma^*, \forall a \in \Sigma, \sigma(s, wa) = \sigma(q, w)\, \sigma(\delta(q, w), a)$$

As before, a string $w \in \Sigma^*$ is *transduced* just in the case that $\delta(s, w) \in F$, and in this case the output is given by $\sigma(s, w)$; that is, $w \mapsto \sigma(s, w)$.


0
f:b
1
o:a
2
o:z
o:r
3

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).

OpenFst and friends

Ancient forebears

- The Xerox toolkit (XFST; Beesley and Karttunen 2003)
- The AT&T toolkit (FSM; Mohri et al. 2000)

Competitors (see Gorman, 2016)

- Carmel (Knight and Graehl, 1998)
- HFST (Lindén et al., 2013)
- Foma (Hulden, 2009)
- Kleene (Beesley, 2012)

OpenFst (Allauzen et al., 2007)

OpenFst is a 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)

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/fst*`
- A Python extension module `pywrapfst`

OpenFst extensions I

`./configure` ...

- `–enable-compress` (Mohri et al., 2015): FST compression
- `–enable-linear-fsts` (Wu et al., 2014): encodes linear models as WFSTs
- `–enable-pdt` (Allauzen and Riley, 2012): pushdown transducer reprsentations and algorithms
- `–enable-ngram-fsts` (Sorensen and Allauzen, 2011): LOUDS compression for n-gram models encoded as WFSAs

OpenGrm I

- 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.

OpenFst conventions I

- 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 operations.
- Both acceptors and transducers, weighted or unweighted, are represented as weighted transducers.
- FST state IDs are integers 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 ϵ 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 naturaly *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 computed destructively (on *A*) using `A.concat(B)` or constructively using `A + B`. The algorithm works by adding an ϵ-arc from every final state in *A* to the initial state of *B*.

Union

The union $A \mid B$ can be computed destructively (on A) using `A.union(B)` or constructively using `A | B`. The algorithm introduces an ϵ-arc from the initial state of A to the initial state of B.

Closure

The closure A^* can be computed destructively using `A.closure()`, or constructively using `closure(A)`. The algorithm introduces ϵ-arcs from all final states to the initial state.

Composition

The composition $A \circ B$ can be computed constructively using `A @ B` or `compose(A, B)`. By default, non-(co)accessible states are trimmed.

Cross-product

The cross-product function `transducer` constructively computes the cross-product transducer $T = A \times B$. It is defined roughly as follows:

```
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")
```

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.

Optimization for unweighted acceptors

```
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()
```

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

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.

Rewrite rule compilation

The context-dependent rewrite rule compilation function `cdrewrite` constructively expands an SPE-like phonological rule specification into a transducer using the Mohri and Sproat (1996) algorithm.

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. Unique paths can be obtained by determinizing the WFST on the fly.

Examples

Rule-based g2p

Consider a simple example: (Latin American, mainland) Spanish grapheme-to-phoneme conversion: `https://gist.github.com/kylebgorman/124909662f1abdab9a97ef06237c557d`

Pair n-gram g2p

Following Novak et al. (2012):

- Train a unigram grapheme-to-phoneme aligner using expectation maximization
- Using the unigram aligner, decode the training data using the shortest-path algorithm to obtain best alignments
- Encode the alignments as an unweighted acceptor
- Train a conventional high-order n-gram model on the encoded alignments
- Decode the alignments to obtain a weighted transducer

A Breakfast Experiment™

- Pronunciations from the Santiago Lexicon (SLR34) of Spanish pronunciations:
    - 73k training words
    - 9k development words
    - 9k test words
- 10 random starts of the aligner (trained with the Viterbi approximation)
- Kneser-Ney smoothing
- N-gram order tuned on the development set (nothing else tuned)
- N-gram model shrunk down to 1m n-grams using relative entropy pruning (Stolcke, 1998)

Results (4-gram model): LER = .0004, WER = .0024.

Speech grammars at Google

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

- Gorman and Sproat (2016) propose an algorithm—implemented in Pynini—which can induce a number grammars from a few-hundred labeled examples.
- Ritchie et al. (2019) describe how Pynini is used to build “unified” verbalization grammars that can be share by both ASR and TTS.

Constrained sequence mapping at Google

- 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.

Some recommended reading

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

More information

http://pynini.opengrm.org

References I

C. Allauzen and M. Riley. A pushdown transducer extension for the OpenFst library. In *CIAA*, pages 66–77, 2012.

C. Allauzen and M. Riley. Algorithms for weighted finite automata with faillure transitions. In *CIAA*, pages 46–58, 2018.

C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: a general and efficient weighted finite-state transducer library. *CIAA*, pages 11–23, 2007.

C. Allauzen, M. Riley, and J. Schalkwyk. Filters for efficient composition of weighted finite-state transducers. In *CIAA*, pages 28–38, 2010.

K. R. Beesley. Kleene, a free and open-source language for finite-state programming. In *10th International Workshop on Finite State Methods and Natural Language Processing*, pages 50–54, 2012.

K. R. Beesley and L. Karttunen. *Finite state morphology*. CSLI, Stanford, CA, 2003.

References II

J. Chandlee, R. Eyraud, and J. Heinz. Learning strictly local subsequential functions. *Transactions of the Association for Computational Linguistics*, 2:491–503, 2014.

J. Chandlee, J. Heinz, and A. Jardine. Input strictly local opaque maps. *Phonology*, 35(2):171–205, 2018.

N. Chomsky and G. A. Miller. Introduction to the formal analysis of natural languages. In R. D. Luce, R. R. Bush, and E. Galanter, editors, *Handbook of mathematical psychology*, pages 269–321. Wiley, New York, 1963.

E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

K. Gorman. Pynini: a Python library for weighted finite-state grammar compilation. In *ACL Workshop on Statistical NLP and Weighted Automata*, pages 75–80, 2016.

References III

K. Gorman and R. Sproat. Minimally supervised number normalization. *Transactions of the Association for Computational Linguistics*, 4: 507–519, 2016.

J. Heinz. Learning long-distance phonotactics. *Linguistic Inquiry*, 41(4): 623–661, 2010.

M. Hulden. Foma: a finite-state compiler and library. In *EACL*, pages 29–32, 2009.

A. Jardine and J. Heinz. Learning tier-based strictly 2-local languages. *Transactions of the Association for Computational Linguistics*, 4: 87–98, 2016.

S. C. Kleene. Representations of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata studies*, pages 3–42. Princeton University Press, Princeton, 1956.

K. Knight and J. Graehl. Machine transliteration. *Computational Linguistics*, 24(4):599–612, 1998.

References IV

K. Lindén, E. Axelson, S. Drobac, S. Hardwick, J. Kuokkala, J. Niemi, T. A. Pirinen, and M. Silfverberg. HFST: a system for creating NLP tools. In *SCFM*, pages 53–71, 2013.

M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3): 321–350, 2002.

M. Mohri. Weighted automata algorithms. In M. Droste, W. Kuich, and H. Vogler, editors, *Handbook of weighted automata*, pages 213–254. Springer, New York, 2009.

M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *ACL*, pages 231–238, 1996.

References V

M. Mohri, F. Pereira, and M. Riley. The design principles and algorithms of a weighted grammar library. *Theoretical Computer Science*, 231(1): 17–32, 2000.

M. Mohri, M. Riley, and A. T. Suresh. Automata and graph compression. In *IEEE International Symposium on Information Theory*, pages 2989–2993, 2015.

A. H. Ng, K. Gorman, and R. Sproat. Minimally supervised written-to-spoken text normalization. In *ASRU*, pages 665–670, 2017.

J. Novak, N. Minematsu, and K. Hirose. WFST-based grapheme-to-phoneme conversion: open-source tools for alignment, model-building and decoding. In *10th Workshop on Finite State Methods and Natural Language Processing*, pages 45–49, 2012.

B. H. Partee, A. ter Meulen, and R. E. Wall. *Mathematical methods in linguistics*. Kluwer, Dordrecht, 1993.

References VI

S. Ritchie, R. Sproat, K. Gorman, D. van Esch, C. Schallhart, N. Bampounis, B. Brard, J. F. Mortensen, M. Holt, and E. Mahon. Unified verbalization for speech recognition & synthesis across languages. In *INTERSPEECH*, pages 3530–3534, 2019.

B. Roark, R. Sproat, C. Allauzen, M. Riley, J. Sorensen, and T. Tai. The OpenGrm open-source finite-state grammar software libraries. In *ACL System Demonstrations*, pages 61–66, 2012.

J. Rogers, J. Heinz, G. Bailey, M. Edlefsen, M. Visscher, D. Wellcome, and S. Wibel. On languages piecewise testable in the strict sense. In *Conference on Mathematics of Language*, pages 255–265, 2010.

J. Sorensen and C. Allauzen. Unary data structures for language models. In *INTERSPEECH*, pages 1425–1428, 2011.

A. Stolcke. Entropy-based pruning of backoff language models. In *DARPA Broadcast News And Understanding Workshop*, pages 270–274, 1998.

References VII

K. Wu, C. Allauzen, K. Hall, M. Riley, and B. Roark. Encoding linear models as weighted finite-state transducers. In *INTERSPEECH*, pages 1258–1262, 2014.

H. Zhang, R. Sproat, A. H. Ng, F. Stahlberg, X. Peng, K. Gorman, and B. Roark. Neural models of text normalization for speech applications. *Computational Linguistics*, 45(2):293–337, 2019.