Finite-state text processing

Kyle Gorman
Graduate Center, City University of New York

About me

- 2003–2006: BA, Linguistics, University of Illinois at Urbana-Champaign
- 2006–2012: PhD, Linguistics, University of Pennsylvania
- 2012–2014: Postdoc, Center for Spoken Language Understanding, Oregon Health & Science University
- 2014–2015: Assistant professor, Center for Spoken Language Understanding, Oregon Health & Science University
- 2015–present: Software engineer, Google
- 2018–present: Assistant professor, Graduate Center, City University of New York

Other stuff: music, exercise, cooking, blogging, TNR

Other things

- I will be at the social events later this week.
- I will be giving a more advanced talk on related materials at the Information Sciences Institute (Marina del Rey) on Thursday.
- Say hi if you're ever in New York, in Midtown East near the Empire State Building and Koreatown.

Learning goals

In this workshop, you will discover:

- finite-state transducers (FSTs) and their connections to:
    - formal language theory and
    - rewrites system;
- and Pynini, a Python library for finite-state development.

Why FSTs?

FSTs are:

- are formally and conceptually simple:
    - they admit proof-theoretic study and
    - polynomial-time algorithms.
- have natural affinities for processing and generating language.
- are one of the main methods for delivering multilingual SLT to users.
- are already in your life:
    - they on your phone and
    - in your browser,
- have proved surprisingly resilient to the “deep learning tsunami” (Manning, 2015).

FSTs at Google

FSTs are used at Google for, e.g.:

- automatic speech recognizers (ASR), particularly embedded low-latency models,
- the front-end of text-to-speech synthesizers (TTS), and
- input method engines (IME) for mobile text entry.

Pynini at Google

Pynini is used extensively at Google for speech technologies, e.g.:

- Gorman and Sproat (2016) describe an algorithm—implemented in Pynini—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 shared by 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.
- Gorman et al. (2021) describe an FST-based noisy channel model for expanding abbreviations in text.

How it’s going to go

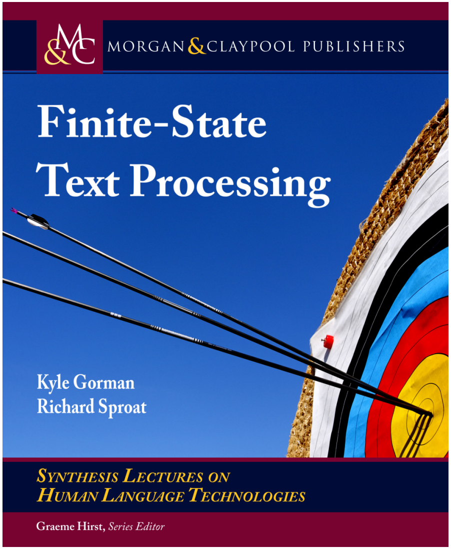
- I’m going to show you math.
- I’m going to show you pictures.
- I’m going to show you Python.

How to learn

- I’m going to show you math:
    - But I’m going to tell you what it means,
    - so it’s okay to sorta let it wash over.
- I’m going to show you pictures:
    - It is important to understand how these are interpreted,
    - though depending on graphics to understand what an FST does can become a crutch.
- I’m going to show you Python:
    - I will type some of the examples into Google Colab (`https://colab.research.google.com/`), and
    - you’re welcome to do the same yourself.

The book

Nearly everything I will say today can be found, in greater detail, in *Finite-State Text Processing* (Gorman and Sproat, 2021). Your library probably has a copy via the Synthesis Digital Library collection. (USC's library does.)


MORGAN & CLAYPOOL PUBLISHERS

Finite-State Text Processing

Kyle Gorman
Richard Sproat

SYNTHESIS LECTURES ON HUMAN LANGUAGE TECHNOLOGIES

Graeme Hirst, *Series Editor*

http://wellformedness.com/courses/fstp/

Outline

- Formal preliminaries
- Finite-state acceptors
- Finite-state transducers
- Rewrite rules

Motivating examples and demos will be weaved throughout. Note that we’ll also take the coffee break at 10:45–11:00.

Formal preliminaries

Why formal languages?

Formal languages are relevant both to the cognitive science of language—by giving a precise definition of the notion “language”, they allow us to, for example, define what it means to *learn* a language—and for speech and language technologies.

Sets

A *set* is an abstract, unordered collection of distinct objects, the *members* or *elements* of that set.

- They are an abstract, logical notion, and we do not presuppose any particular method of representing them in hardware or software.
- They are unordered in the sense that we do not need there to be any natural order among the elements or members of a set.
- Sets may either be finite (e.g., the set consisting of students in this class) or infinite (e.g., the set of grammatical sentences of English).

Set membership

- Members of a set can be of any type, including other sets.
- Set membership is indicated with the $\in$ symbol.
    - The expression $x \in X$ is read "x is a member of X".
- We can also deny this relation using $\notin$.
    - The expression $x \notin X$ is read "x is not a member of X".

Subsets

- The 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. We indicate this using $\subseteq$.
    - The expressions $X \subseteq Y$ is read “X is a subset of Y”.
- We can also deny this relation using $\nsubseteq$.
    - The expression $X \nsubseteq Y$ is read “X is not a subset of Y”.

The empty set

The set with no members is known as the *empty set*. It is written as $\varnothing$ (i.e., rather than an empty set of curly braces).

Problem

How are sets, as defined here, like Python `set` objects? How are they different?

Solution

- Like Python sets, our sets are unordered.
- However, objects stored in a Python set must be immutable and hashable. (Because of this restriction, Python sets may not contain other Python sets.)
- Python sets may not be infinite.

Set notation

By convention, we use capital Italic letters (e.g., *X*, *Y*, *Z*) to denote sets, and lowercase Italic letters (e.g., *x*, *y*, *z*) to denote members. There are then two ways to specify the contents of a set. . . .

Extension (or list) notation

For finite sets, we can simply list the extension of the set, enclosed in curly braces.

$$\{2, 3, 5, 7\}$$

Note that it is an accidental feature that the members of a set are listed in a particular order; there is no natural ordering of the members of a set. Thus all the following are equivalent.

$$\{2, 3, 5, 7\}, \{7, 5, 3, 2\}, \{3, 2, 7, 5\}, \{2, 5, 3, 7\}, \ldots$$

Predicate (or set-builder) notation

Alterantively (and necessarily, for infinite sets), we can intensionally describe properties that uniquely identify the set's members.

$$\{x \mid 13 \leq x \leq 27\}$$

Problem

Let:

$$K = \{\text{Mars, Saturn, Uranus}\}$$
$$L = \{x \mid x \text{ is a planet in our solar system}\}$$

- Is K a member of L?
- And, K a subset of L?

Solution

- $K \notin L$
- $K \subseteq L$

Set operations

Sets support several elementary logical operations including *union*, *intersection*, and *difference*.

Union

The *union* of two sets $X \cup Y$ is the set that contains just the elements which are members of X, of Y, or both X and Y. It corresponds to *disjunction operator* $\vee$ in logic, and (loosely) to the conjunction *or* in English.

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

Intersection

The *intersection* of two sets $X \cap Y$ is the set that contains just the elements which are members of both X and Y. It corresponds to the *conjunction operator* $\wedge$ in logic, and to the conjunction *and* in English.

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

Difference

The *difference* of two sets $X - Y$ is the set that contains just the elements which are members of X but not members of Y.

$$X - Y = \{x \mid x \in X \land x \notin Y\}$$

Problem

Let:

$$K = \{a, b\}$$
$$L = \{c, d\}$$
$$M = \{b, d\}$$

$$K \cup L = \_\_$$
$$L \cup M = \_\_$$
$$K \cap L = \_\_$$
$$K \cap M = \_\_$$
$$L \cap M = \_\_$$
$$K - M = \_\_$$
$$M - L = \_\_$$

Solution

$$\begin{aligned}
K \cup L &= \{a, b, c, d\} \\
L \cup M &= \{b, c, d\} \\
K \cap L &= \varnothing \\
K \cap M &= \{b\} \\
L \cap M &= \{d\} \\
K - M &= \{a\} \\
M - L &= \{b\}
\end{aligned}$$

Closure

Let $\bullet$ be a binary (infix) operator, and let Z be a set. Then, Z is said to be *closed with respect to* (or *have closure over*) $\bullet$ if for all subsets X and Y of Z, $X \bullet Y \in Z$.

Closure properties of sets

Sets are closed with respect to union, intersection, and difference, among other operators.

Pairs

A *pair* or *two-tuple* is a sequence of two elements; e.g., (a, b) is the pair consisting of a then b.

Cross-product

The *cross-product* (or *Cartesian product*) of two sets, $X \times Y$, is the set that contains all pairs (x, y) where x is an element of X and y is an element of Y.

$$X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$$

Relations

A (*two-way* or *binary*) *relation* over sets X and Y is a subset of the cross-product $X \times Y$.

Relation notation

By convention, lowercase Greek letters indicate relations. For binary relations, the *domain*—set of inputs—and *range*—the set of outputs—are usually provided upon first definition. For example, the "less than" relation might be written $\lambda \subseteq \mathbb{R} \times \mathbb{R} = \{(x, y) \mid x < y\}$ where $\mathbb{R}$ is the set of real numbers.

N-ary relations

Three-, four- and five-way relations, and so on, are also well-defined.

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 monospaced (or typewriter) text is used to denote string literals.

The empty string

The null string with is known as the *empty string*. It is written ϵ ("epsilon").

Concatenation

The *concatenation* of two strings *s* and *t*, written *st*, is the string defined by the sequence of symbols in *s* followed by the sequence of symbols in *t* end-to-end.

Reversal

The *reversal* of a string s, written s^R, is the string defined by the sequence of symbols in s in reverse order.

Problem

Let:

$$s = \texttt{aab}$$
$$t = \texttt{cdf}$$

$$s^R t = \_\_$$
$$(st)^R = \_\_$$

Solution

$$\begin{aligned} s^R t &= \texttt{baacdf} \\ (st)^R &= \texttt{fdcbaa} \end{aligned}$$

Languages

Sets of strings are called *languages*. This is merely a term of art; it is not intended to supplant the common-sense notions of what a language is.

Concatenation (again)

If X and Y are languages, then XY contains the concatenation of each string $x \in X$ with each string $y \in Y$.

$$X\,Y = \{x\,y \mid x \in X \wedge y \in Y\}$$

Range concatenation

The notation X^n, where n is a natural number, denotes a language consisting of n “self-concatenations” of X.

$$X^0 = \{\epsilon\}$$

$$X^4 = XXXX$$

Kleene star

The *closure* of a language *X* is the infinite union of zero or more concatenations of *X* with itself. It is denoted by a superscripted asterisk.

$$\begin{aligned} X^* &= \bigcup_{i \geq 0} X^i \\ &= \{\epsilon\} \cup X \cup XX \cup XXX \cup \ldots \end{aligned}$$

Kleene plus

A variant of closure, denoted by a superscripted plus-sign, excludes the empty string.

$$\begin{aligned} X^+ &= \bigcup_{i>0} X^i \\ &= X \cup XX \cup XXX \cup \ldots \\ &= XX^* \end{aligned}$$

Kleene question mark

A superscripted question mark indicates optionality.

$$X^{?} = \{\epsilon\} \cup X$$

Closure properties of languages

Languages are closed with respect to:

- union, intersection, difference,
- concatenation, closure, and reversal.

Regular languages

- 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 XY is a regular language, and
    - their union $X \cup Y$ is a regular language.
- Other languages are not regular languages.

Regular expressions

Regular expressions are a terse representation of regular languages which use closure, union, and concatenation. **?**, 17f. describe regular expressions as “unsung successes in standardization in computer science”. Regular expression matching is supported by Python’s `re` module, command-line tools like `grep` and `sed`, and nearly all of these use roughly the same terse algebraic notation. By convention, we write regular expressions in monospaced font, surrounded by forward slashes.

Correspondences: concatenation

- Concatenation is implicit in regular expressions.

$$\texttt{/ab/} = \{\texttt{ab}\}$$

Correspondences: quantifiers

- Kleene star corresponds to the quantifier `*`.

$$\texttt{/a*(bb)*/} = \{\texttt{a}\}^*\{\texttt{bb}\}^*$$

- Kleene plus corresponds to the quantifier `+`.

$$\begin{aligned}\texttt{/yes+/} &= \{\texttt{ye}\}\{\texttt{s}\}^+\\ &= \{\texttt{yes}, \texttt{yess}, \texttt{yesss}, \ldots\}\end{aligned}$$

- The “Kleene question mark” corresponds to the quantifier `?`.

$$\begin{aligned}\texttt{/colou?r/} &= \{\texttt{colo}\}\{\texttt{u}\}^?\{\texttt{r}\}\\ &= \{\texttt{color}, \texttt{colour}\}\end{aligned}$$

Correspondences: union

- Union $\cup$ corresponds to several notations:
  - Square brackets indicate the union of single characters.

$$\begin{aligned}\texttt{/[Dd]addy/} &= (\{\texttt{D}\} \cup \{\texttt{d}\})\{\texttt{addy}\} \\ &= \{\texttt{Daddy}, \texttt{daddy}\}\end{aligned}$$

  - Square brackets can also be used to indicate a union of a range of single characters.

$$\begin{aligned}\texttt{/Rocky\_[1-3]/} &= \{\texttt{Rocky\_}\}(\{\texttt{1}\} \cup \{\texttt{2}\} \cup \{\texttt{3}\}) \\ &= \{\texttt{Rocky\_1}, \texttt{Rocky\_2}, \texttt{Rocky\_3}\}\end{aligned}$$

  - The | operator indicates unions of arbitrary-length character sequences.

$$\begin{aligned}\texttt{/gupp(y|ies)/} &= \{\texttt{gupp}\}(\{\texttt{y}\} \cup \{\texttt{ies}\}) \\ &= \{\texttt{guppy}, \texttt{guppies}\}\end{aligned}$$

Finite-state acceptors

Kleene 1956

Initially, the study of

- abstract computational devices known as *state machines* and
- formal languages

were considered independent of one another. Kleene (1956) was one of the first to unify these two areas of study. Kleene wished to characterize the properties of *nerve nets* (McCulloch and Pitts, 1943), a primitive form of artificial neural network. In doing so, Kleene introduced the regular languages and formalized the connection between regular languages and *finite acceptors*, a type of state machine.

Regular languages in the 20th century

- Regular languages were popularized in part by discussion of the *Chomsky*(-*Schützenberger*) hierarchy (e.g., Chomsky and Miller, 1963).
- Regular languages were used by Thompson (1968) to create the `grep` regular expression matching utility.
- Finite acceptors are used to compactly store morphological dictionaries.
- Finite acceptors are used to compactly represent *language models*, particularly in speech recognition engines.

It now seems that an enormous amount of linguistically-interesting phenomena can be described in terms of regular languages and the closely-related *rational relations*.

Negative results

At the same time, there were two important negative results:

- Syntactic grammars 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” from positive data under Gold’s (1967) notion of *language identification in the limit*.

In practice, this means that regular languages and finite acceptors are somewhat limited as models of syntax, though they are still well-suited as models of phonology and morphology.

State machines

A *state machine* is hardware or software whose behavior can be described solely in terms of a set of *states* and *arcs*, transitions between those states. In this formalism, states roughly correspond to “memory” and arcs to “operations” or “computations”. A *finite-state machine* is merely a state machine with a finite, predetermined set of states and labeled arcs.

As directed graphs

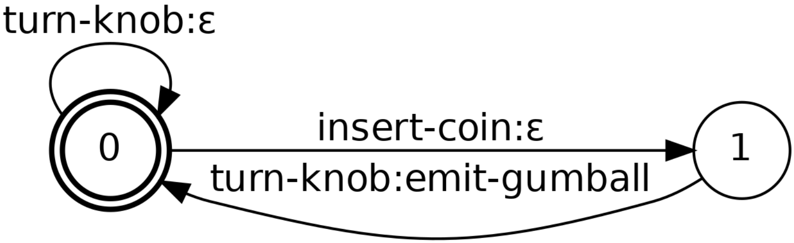
State machines are examples of what computer scientists call *directed graphs*. These are “directed” in the sense that the existence of an arc from state *q* to state *r* does not imply an arc from *r* to *q*. In *state diagrams*, we indicate this directionality using arrows.

(image: credit: Wikimedia Commons)

The humble gumball machine

One familiar example of a state machine—encoded in hardware, rather than software—is the old-fashioned gumball machine. Each state of the gumball machine is associated with actions such as

- turning the knob,
- inserting a coin, or
- emitting a gumball.


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

Application

The application of an input argument to a relation is indicated using square brackets. For instance given the successor function σ, then $\sigma[3] = \{4\}$ because $(3, 4) \in \sigma$.

Finite-state acceptors

An *finite-state acceptor* (FSA) is a 5-tuple defined by:

- a finite set of states Q,
- a *start* or *initial* state $s \in Q$,
- a set of *final* or *accepting* $F \subseteq Q$,
- an *alphabet* Σ, and
- a *transition relation* $\delta \subseteq Q \times \left(\Sigma \cup \{\epsilon\}\right) \times Q$.

Note that, as formalized here, there is only one start state but may be multiple final states, and that the start state may also be a final state.

Acceptance

An FSA is said to *accept*, *match*, or *recognize* a string if there exists a path from the initial state to some final state, and the labels of the arcs traversed by that state correspond to the string in question. The set of all strings so accepted are called the FSA's language.

Paths

Given two states $q, r \in Q$ and a symbol $z \in \Sigma \cup \{\epsilon\}$, $(q, z, r) \in \delta$ implies that there is an arc from state q to state r with label z. A *path* through a finite acceptor is a pair of

- a state sequence $q_1, q_2, \ldots, q_n \in Q^n$ and a
- a string $z_1, z_2, \ldots, z_n \in (\Sigma \cup \{\epsilon\})^n$,

subject to the constraint that $\forall i \in [1, n] : (q_i, z_i, q_{i+1}) \in \delta$; that is, there exists an arc from q_i to q_{i+1} labeled z_i.

Complete paths

A path is said to be *complete* if

- $(s, z_1, q_1) \in \delta$ and
- $q_n \in F$.

That is, a complete path must also begin with an arc from the initial state s to q_1 labeled z_1 and terminate at a final state. Then, an FSA accepts string $z \in (\Sigma \cup \{\epsilon\})^*$ if there exists a complete path with string z.

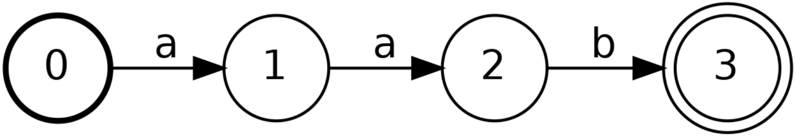
Kleene’s theorem

Kleene’s theorem holds that any regular language is accepted by an FSA, and any language accepted by an FSA is a regular language. This implies that because regular languages are closed under closure, concatenation, and union, so are FSAs.

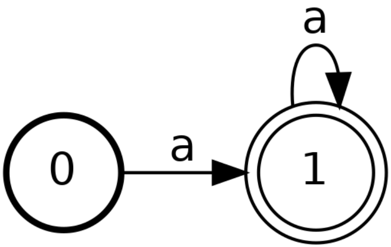
Reading the state diagrams

- States are indicated by circles.
- The initial state is indicated by a bold circle.
- Final states are indicated by double-struck circles.
- Labeled arrows indicate arcs.

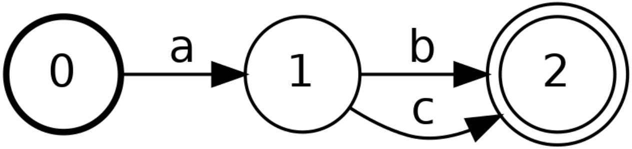
{aab}



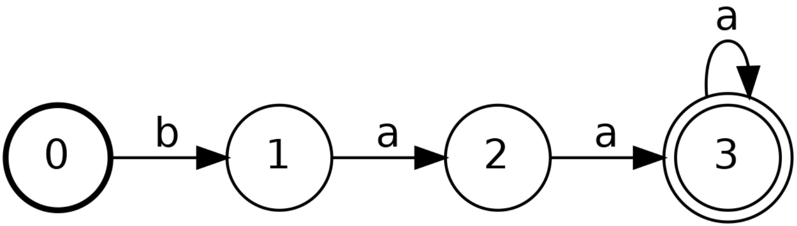
$\{a\}^+$



$\{a\}(\{b\} \cup \{c\})$



$\{ba\}\{a\}^+$



The sheep language

- $Q = \{0, 1, 2, 3\}$
- $s = 0$
- $F = \{3\}$
- $\Sigma = \{\mathrm{a}, \mathrm{b}\}$
- $\delta = \{(0, \mathrm{b}, 1), (1, \mathrm{a}, 2), (2, \mathrm{a}, 3), (3, \mathrm{a}, 3)\}$

ϵ equivalence

Every ϵ-FSA has an equivalent ϵ-free (or “e-free”) FSA that can be found using the epsilon-removal algorithm (Mohri, 2002a).

Demo

Finite-state transducers

Cross-product (redux) and rational relations

Recall that a *cross-product* (or *Cartesian product*) of two sets, $X \times Y$, is the set that contains all pairs (x, y) where x is an element of X and y is an element of Y.

$$X \times Y = \{(x, y) \mid x \in X \land y \in Y\}$$

Then, a rational relation is a subset of the cross-product of two regular languages (e.g., $\gamma \subseteq A \times B$).

Example: state abbreviations

$$\begin{aligned}\gamma = \{&(\mathrm{AK}, \texttt{Alaska}),\\ &(\mathrm{AL}, \texttt{Alabama},\\ &(\mathrm{AR}, \texttt{Arkansas}),\\ &(\mathrm{AZ}, \texttt{Arizona}),\\ &(\mathrm{CA}, \texttt{California}),\\ &(\mathrm{CO}, \texttt{Colorado}),\\ &(\mathrm{CT}, \texttt{Connecticut}),\\ &(\mathrm{DE}, \texttt{Delaware}),\\ &\ldots\}\end{aligned}$$

Interpretation

Regular languages are *languages*, or sets of strings. Rational relations, in turn, can either be thought of as

- sets of pair of (input and output) strings, or as
- mappings between input and output strings.

Thus, we might say either that

- $(\texttt{OH}, \texttt{Ohio}) \in \gamma$, or
- $\gamma[\{\texttt{OH}\}] = \{\texttt{Ohio}\}$.

Finite-state transducers

Finite-state transducers (FSTs) are generalizations of finite-state acceptors which correspond to the rational relations. An FST is a 6-tuple defined by

- a finite set of states Q,
- a *start* or *initial* state $s \in Q$,
- a set of *final* or *accepting* states $F \subseteq Q$,
- an ***input alphabet*** Σ,
- an ***output alphabet*** Φ, and
- a ***transition relation*** $\delta \subseteq Q \times \left(\Sigma \cup \{\epsilon\}\right) \times \left(\Phi \cup \{\epsilon\}\right) \times Q$.

Transduction

An FST is said to *transduce* or *map* from $x \in (\Sigma \cup \{\epsilon\})^*$ to $y \in (\Phi \cup \{\epsilon\})^*$ so long as a complete path with input string x and output string y exists.

Paths

Given two states $q, r \in Q$, input symbol $x_i \in \Sigma \cup \{\epsilon\}$, and output symbol $y_i \in \Phi \cup \{\epsilon\}$, $(q, x_i, y_i, r) \in \delta$ implies that there is an arc from state q to state r with input label x_i and output label y_i. A *path* through a finite transducer is a triple consisting of

- a state sequence $q_1, q_2, q_3, \ldots \in Q^n$ and a
- a input string $x_1, x_2, x_3, \ldots \in (\Sigma \cup \{\epsilon\})^n$,
- a output string $y_1, y_2, y_3, \ldots \in (\Phi \cup \{\epsilon\})^n$,

subject to the constraint that $\forall i \in [1, n] : (q_i, x_{i+1}, y_{i+1}, q_{i+1}) \in \delta$; that is, there exists an arc from q_i to q_{i+1} labeled $x_{i+1} : y_{i+1}$.

Complete paths

A path is said to be *complete* if

- $(s, x_1, y_1, q_1) \in \delta$ and
- $q_n \in F$.

A complete path must also begin with an arc from the initial state s to q_1 labeled $x_1 : y_1$ and terminate at a final state. Then, an FST transduces input string $x \in (\Sigma \cup \{\epsilon\})^*$ to output string $Y \in (\Phi \cup \{\epsilon\})^*$ if there exists a complete path with input string x and output string y.

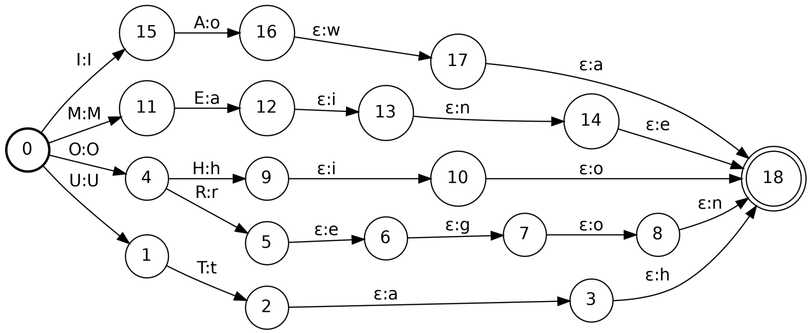
FSAs as FSTs

FSAs can be thought of as a special case of FSTs where every transition has the same input and output label. This is why, in Pynini and friends, FSAs are instance of a class called `Fst`.

Why ϵs

FSTs can map between strings of different lengths, but one must use ϵs to "pad out" the shorter string. Thus, whereas every FSA has an equivalent "e-free" FSA, not all ϵ-FSTs have an equivalent "e-free" form.

State abbreviations (fragment)



Rational operations over FSTs

Rational relations—and thus FSTs—are closed under closure, concatenation, and union, and the Thompson (1968) constructions for these operations are also appropriate to FSTs.

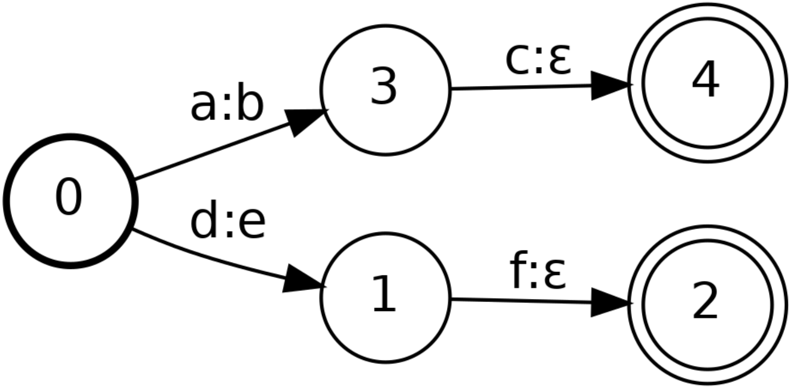
Projection

Projection converts a FST to an FSA that is either equal to its domain (*input-projection*) or range (*output-projection*). By convention, input-projection is indicated by the prefix operator π_i and output-project by π_o. Projection can be computed simply by copying all input (resp. output) labels onto the ouput (resp. input) labels along each arc.

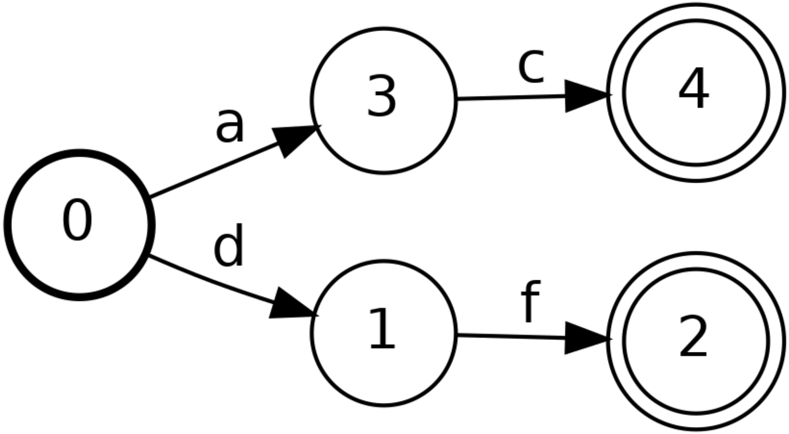
Inversion

Inversion swaps the domain and range of an FST. By convention, it is indicated by a superscripted -1. Inversion can be computed simply by swapping input and output labels along each arc.

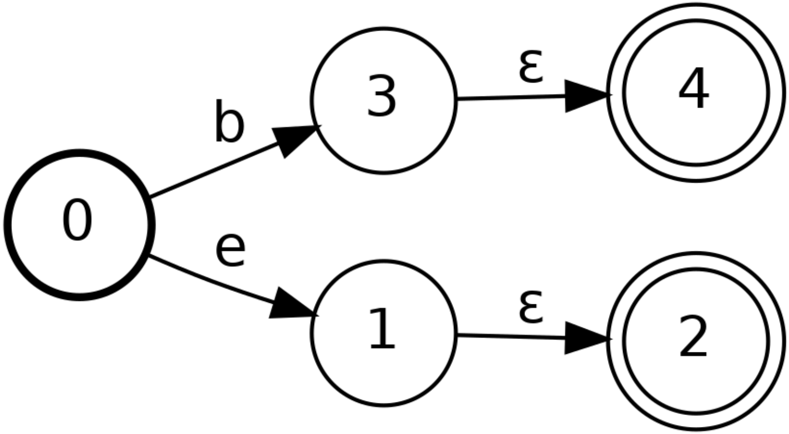
$(\{\mathtt{ac}\} \times \{\mathtt{b}\}) \cup (\{\mathtt{df}\} \times \{\mathtt{e}\})$



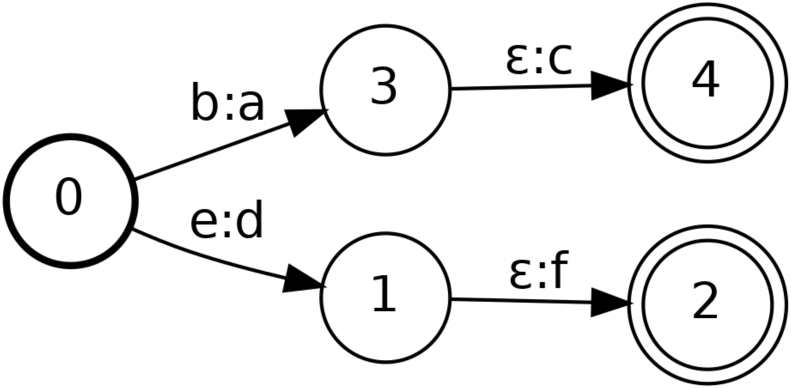
$\pi_i\,((\{\mathtt{ac}\} \times \{\mathtt{b}\}) \cup (\{\mathtt{df}\} \times \{\mathtt{e}\}))$



$\pi_o((\{\texttt{ac}\} \times \{\texttt{b}\}) \cup (\{\texttt{df}\} \times \{\texttt{e}\}))$



$$((\{\mathtt{ac}\} \times \{\mathtt{b}\}) \cup (\{\mathtt{df}\} \times \{\mathtt{e}\}))^{-1}$$



Intersection

Recall that the regular languages—and thus FSAs—are closed under intersection. However, FSTs are not closed under intersection.

Composition

Composition is a generalization of intersection and relation chaining. Its precise interpretation depends on whether the inputs are languages/FSAs M, N or relations/FSTs μ, ν:

- $M \circ N$ yields their intersection $M \cap N$.
- $M \circ \nu$ yields $\{(a, b) \mid a \in M \wedge b \in \nu[a]\}$; i.e., it restricts the domain of ν by intersecting it with M.
- $\mu \circ N$ yields $\{(a, b) \mid b \in \mu[a] \wedge b \in N\}$; i.e., it restricts the range of μ by intersecting it with N.
- $\mu \circ \nu$ yields $\{(a, c) \mid b \in \mu[a] \wedge c \in \nu[b]\}$; i.e., it chains the output of μ to the input of ν.

Associativity

Composition is associative and *n*-ary composition can be implemented by a sequence of two-way compositions. Note however that for automata, one bracketing into a sequence of two-way compositions—e.g., $A \circ B \circ C$ factored as the *left-associative* $(A \circ B) \circ C$ versus the *right-associative* $A \circ (B \circ C)$—may be far more efficient than other equivalent associativities.

Demo

Briefly noted:
Weights
Shortest distance
Shortest path

Rewrites

Why rewrites?

- Grammarians, since at least Pāṇini (fl. 4th c. BCE), have conceived of grammars not as sets of permissible strings but rather as a series of rules which “rewrite” abstract inputs to produce surface forms.
- One particularly influential rule notation is the one popularized by Chomsky and Halle (1968), henceforth SPE.
- Johnson (1972) proves this notation, with some sensible restrictions, is equivalent to the *rational relations* and thus to *finite transducers*.

Formalism

Let Σ be the set of symbols over which the rule will operate.

- For phonological rules, Σ might consist of all phonemes and their allophones in a given language.
- For grapheme-to-phoneme rules, it would contain both graphemes and phonemes.

Let $s, t, l, r \in \Sigma^*$. Then, the following is a possible rewrite rule.

$$s \rightarrow t \;/\; l \_\_ r$$

where $s \rightarrow t$ is the *structural change* and l and r as the *environment*. By convention, l and/or r can be omitted when they are null (i.e., are ϵ).

Interpretation

The above rule can be read as “*s* goes to *t* between *l* and *r*”, and specifies a rational relation with domain and range Σ^* such that all instances of *lsr* are replaced with *ltr*, with all other strings in Σ^* passed through.

Example

Let $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and consider the following rule.

$$\mathsf{b} \rightarrow \mathsf{a} \ / \ \mathsf{b} \_\_ \mathsf{b}$$

$$\begin{array}{lcl} \mathsf{bbba} & \rightarrow & \mathsf{baba} \\ \mathsf{abbbabbbc} & \rightarrow & \mathsf{ababababc} \end{array}$$

Input: `cbbca`

Output: `cbbca`

Input: abbbba

Output: ???

Directionality

However, application is ambiguous with respect to certain input strings.

a.	*simultaneous* application	`abaaba`
b.	*left-to-right* or *right-linear* application	`ababba`
c.	*right-to-left* or *left-linear* application	`abbaba`

Directional application

In SPE it is assumed that that all rules apply simultaneously (op. cit., 343f.). However, Johnson (1972) adduces a number of phonological examples where directional application—either left-to-right or right-to-left—is required. However, note that directionality has no discernable effect on many rules and can often be ignored.

Boundary symbols

Let $\hat{}, \$ \notin \Sigma$ be *boundary symbols* disjoint from Σ. Now let $\hat{}$, the beginning-of-string symbol, to optionally appear as the leftmost symbol in *l*, and permit $, the end-of-string-symbol, to optionally appear as the rightmost symbol in *r*. These boundary symbols are not permitted to appear elsewhere in *l* or *r*, or anywhere within the structural description and change.

Example

Let $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ and consider the following rule.

$$\mathsf{b} \rightarrow \mathsf{a} \;/\; \hat{}\;\mathsf{b}\,\_\_\,\mathsf{b}$$

$$\begin{array}{lcl} \mathsf{bbba} & \rightarrow & \mathsf{baba} \\ \mathsf{abbbc} & \rightarrow & \mathsf{abbbc} \end{array}$$

Generalization

We can generalize the elements of rules from single strings to languages and relations. Then, a rewrite rule is specified by a five-tuple consisting of

- an *alphabet* Σ,
- a *structural change* $\tau \subseteq \Sigma^* \times \Sigma^*$,
- a *left environment* $L \subseteq \{\hat{}\}^? \Sigma^*$,
- a *right environment* $R \subseteq \Sigma^* \{\$\}^?$, and
- a *directionality* (one of: “simultaneous”, “left-to-right”, or “right-to-left”).

Briefly noted:
Features
Abbreviatory devices
Constraint-based formalisms

Rule compilation

Rules which apply at the end or beginning of a string are generally trivial to express as a finite transducer. For example, the following rules prepend a prefix *p* or append a suffix *s*, respectively.

$$\begin{array}{ccl} \varnothing & \rightarrow & \{p\} \ / \ \hat{}\ \_\_\ \Sigma^* \\ \varnothing & \rightarrow & \{s\} \ / \ \Sigma^*\ \_\_\ \$ \end{array}$$

Such rules, respectively, correspond to the rational relations:

$$\left(\{\epsilon\} \times \{p\}\right) \Sigma^*$$
$$\Sigma^* \left(\{\epsilon\} \times \{s\}\right)$$

Challenges

Greater difficulties arise from the possibility of

- multiple sites for application and
- multiple overlapping contexts for application.

It thus proved challenging to develop a general-purpose algorithm for compilation, and was not widely-known until the 1990s (e.g., Kaplan and Kay, 1994; Karttunen, 1995). The following is a generalization put forth by Mohri and Sproat (1996), which builds a rewrite rule from a cascade of five transducers, each a simpler rational relation.

The algorithm I

If X is a language, let $\bar{X}$ denote its *complement*, the language consisting of all strings not in X. Then, let $<_1, <_2, > \notin \Sigma$ be *marker symbols* disjoint from the alphabet Σ. L and R are acceptors defining the left and right contexts, respectively. The constituent transducers are as follows:

- ρ inserts the $>$ marker before all substrings matching R: $\Sigma^* R \rightarrow \Sigma^* > R$.
- ϕ inserts markers $<_1$ and $<_2$ before all substrings matching $\pi_i(\tau) >$: $(\Sigma \cup \{>\})^* \pi_i(\tau) \rightarrow (\Sigma \cup \{>\})^* \{<_1, <_2\} \pi_i(\tau)$. Note that this introduces two paths, one with $<_1$ and one with $<_2$, which will ultimately correspond, respectively, to the cases where L does/does not occur to the left (see steps 4, 5 below).
- γ applies the structural change τ anywhere $\pi_i(\tau)$, the input projection of τ, is preceded by $<_1$ and followed by $>$. It simultaneously deletes the $>$ marker everywhere.

The algorithm II

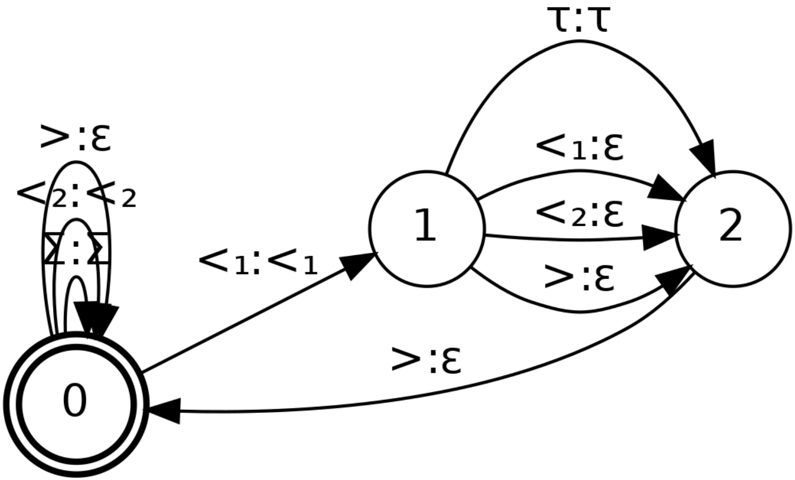
- λ_1 admits only those strings in which L is followed by the $<_1$ marker and deletes all $<_1$ markers satisfying this condition: $\Sigma^* L <_1 \rightarrow \Sigma^* L$.
- λ_2 admits only those strings in which all $<_2$ markers are not preceded by L and deletes all $<_2$ markers satisfying this condition: $\Sigma^* \overline{L} <_2 \rightarrow \Sigma^* \overline{L}$

Then, the final context-dependent rewrite rule transducer is given by

$$T = \rho \circ \phi \circ \gamma \circ \lambda_1 \circ \lambda_2$$

Slight variants are used for right-to-left and simultaneous transduction.

Schematic of γ



Briefly noted:
Efficiency considerations

Demo

Appendix A: further reading

Further reading

- Mohri, 2009: reviews major WFST algorithms
- Roark and Sproat, 2007, §1: introduces WFST weights
- Mohri, 2002b: introduces shortest-distance and shortest-path algorithms

More information

openfst.org

opengrm.org

baumwelch.opengrm.org

ngram.opengrm.org

pynini.opengrm.org

thrax.opengrm.org

Appendix B: Pynini installation

Pynini installation

Pynini can be run on most modern UNIX-like operation systems, including MacOS, Linux, or on Windows, using the Windows Subsystem for Linux (WSL). For most users, the simplest option is to install the module and its dependencies using Anaconda, or to use it via Colab.

Anaconda installation

- Install either Anaconda or Miniconda for your platform. (Note that if you are planning on using Pynini on Windows, you should download and run the Linux installer and run it from the WSL terminal.)
- At the command line, issue the following command:

```
conda install -c conda-forge pynini
```

Colab installation

If you wish to just use Pynini in a Colab notebook, add the following to the top of your notebook:

```
!pip install pynini
%load_ext wurlitzer
```

Appendix C: OpenFst & OpenGrm

OpenFst (Allauzen et al., 2007)

OpenFst is a open-source C++17 library for weighted finite state transducers developed at Google.

- One serialization format (`.fst`) is shared across all OpenFst and OpenGrm libraries.
- FSTs can be compacted; e.g., unweighted string acceptors can be stored as integer arrays.
- Collections of FSTs can be stored in FST archives (`.far`), a shardable key-value store.

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`

OpenGrm

- Baum-Welch (Gorman et al., 2021): 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
- Pynini (Gorman, 2016; Gorman and Sproat, 2021): Python extension module for WFST grammar development
- SFst (Allauzen and Riley, 2018): CLI tools and libraries for building *stochastic FSTs*
- Thrax (Roark et al., 2012): DSL-based compiler for WFST grammar development

References I

C. Allauzen and M. Riley. Algorithms for weighted finite automata with failure transitions. In *Proceedings of the 23rd International Conference on Implementation and Application of Automata*, 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. In *Proceedings of the 12th International Conference on Implementation and Application of Automata*, pages 11–23, 2007.

N. Chomsky and M. Halle. *Sound Pattern of English*. Harper & Row, 1968.

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, 1963.

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

References II

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.

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

K. Gorman and R. Sproat. *Finite-State Text Processing*. Morgan & Claypool, 2021.

K. Gorman, C. Kirov, B. Roark, and R. Sproat. Structured abbreviation expansion in context. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 995–1005, 2021.

C. D. Johnson. *Formal Aspects of Phonological Description*. Mouton, 1972.

R. Kaplan and M. Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, 1994.

References III

L. Karttunen. The replace operator. In *33rd Annual Meeting of the Association for Computational Linguistics*, pages 16–23, 1995.

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, 1956.

C. D. Manning. Last words: computational linguistics and deep learning. *Computational Linguistics*, 41(4):701–707, 2015.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

M. Mohri. Generic epsilon-removal and input epsilon-normalization algorithms for weighted transducers. *International Journal of Computer Science*, 13(1):129–143, 2002a.

References IV

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

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

M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 231–238, 1996.

A. H. Ng, K. Gorman, and R. Sproat. Minimally supervised written-to-spoken text normalization. In *IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 665–670, 2017.

References V

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 *Proceedings of INTERSPEECH*, pages 3530–3534, 2019.

B. Roark and R. Sproat. *Computational Approaches to Morphology and Syntax*. Cambridge University Press, 2007.

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

K. Thompson. Programming techniques: regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

References VI

K. Vijay-Shanker, D. J. Weir, and A. K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Annual Meeting of the Association for Computational Linguistics*, pages 104–111, 1987.

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.