Appeared in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL 2002)*, pp. 1-8, 2002.

Parameter Estimation for Probabilistic Finite-State Transducers*

Jason Eisner
Department of Computer Science
Johns Hopkins University
Baltimore, MD, USA 21218-2691
jason@cs.jhu.edu

Abstract

Weighted finite-state transducers suffer from the lack of a training algorithm. Training is even harder for transducers that have been assembled via finite-state operations such as composition, minimization, union, concatenation, and closure, as this yields tricky parameter tying. We formulate a "parameterized FST" paradigm and give training algorithms for it, including a general bookkeeping trick ("expectation semirings") that cleanly and efficiently computes expectations and gradients.

1 Background and Motivation

Rational relations on strings have become widespread in language and speech engineering (Roche and Schabes, 1997). Despite bounded memory they are well-suited to describe many linguistic and textual processes, either exactly or approximately.

A **relation** is a set of $(input, output)$ pairs. Relations are more general than functions because they may pair a given input string with more or fewer than one output string.

The class of so-called **rational** relations admits a nice declarative programming paradigm. Source code describing the relation (a **regular expression**) is compiled into efficient object code (in the form of a 2-tape automaton called a **finite-state transducer**). The object code can even be optimized for runtime and code size (via algorithms such as determinization and minimization of transducers).

This programming paradigm supports efficient nondeterminism, including parallel processing over infinite sets of input strings, and even allows "reverse" computation from output to input. Its unusual flexibility for the practiced programmer stems from the many operations under which rational relations are closed. It is common to define further useful operations (as macros), which modify existing relations not by editing their source code but simply by operating on them "from outside."

The entire paradigm has been generalized to **weighted relations**, which assign a weight to each $(input, output)$ pair rather than simply including or excluding it. If these weights represent probabilities $P(input, output)$ or $P(output \mid input)$, the weighted relation is called a **joint** or **conditional (probabilistic) relation** and constitutes a statistical model. Such models can be efficiently restricted, manipulated or combined using rational operations as before. An artificial example will appear in §2.

The availability of toolkits for this weighted case (Mohri et al., 1998; van Noord and Gerdemann, 2001) promises to unify much of statistical NLP. Such tools make it easy to run most current approaches to statistical *markup*, *chunking*, *normalization*, *segmentation*, *alignment*, and noisy-channel *decoding*,[1] including classic models for speech recognition (Pereira and Riley, 1997) and machine translation (Knight and Al-Onaizan, 1998). Moreover, once the models are expressed in the finite-state framework, it is easy to use operators to tweak them, to apply them to speech lattices or other sets, and to combine them with linguistic resources.

Unfortunately, there is a stumbling block: *Where do the weights come from?* After all, statistical models require supervised or unsupervised training. Currently, finite-state practitioners derive weights using exogenous training methods, then patch them onto transducer arcs. Not only do these methods require additional programming outside the toolkit, but they are limited to particular *kinds* of models and training regimens. For example, the forward-backward algorithm (Baum, 1972) trains only Hidden Markov Models, while (Ristad and Yianilos, 1996) trains only stochastic edit distance.

In short, current finite-state toolkits include no training algorithms, because none exist for the large space of statistical models that the toolkits can in principle describe and run.

[*] A brief version of this work, with some additional material, first appeared as (Eisner, 2001a). A leisurely journal-length version with more details has been prepared and is available.

[1] Given $output$, find $input$ to maximize $P(input, output)$.

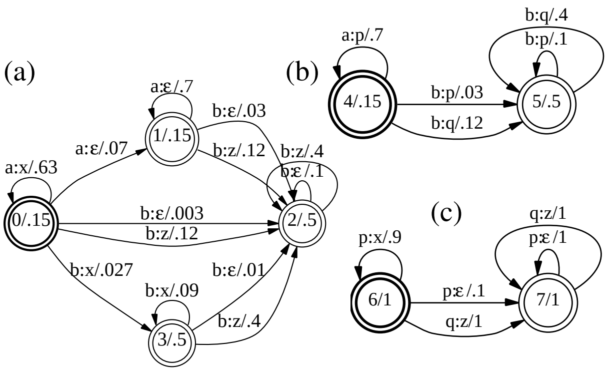

Figure 1: (a) A probabilistic FST defining a joint probability distribution. (b) A smaller joint distribution. (c) A conditional distribution. Defining (a)=(b)∘(c) means that the weights in (a) can be altered by adjusting the fewer weights in (b) and (c).

This paper aims to provide a remedy through a new paradigm, which we call **parameterized finite-state machines**. It lays out a fully general approach for training the weights of weighted rational relations. First §2 considers how to parameterize such models, so that weights are defined in terms of underlying parameters to be learned. §3 asks what it means to learn these parameters from training data (what is to be optimized?), and notes the apparently formidable bookkeeping involved. §4 cuts through the difficulty with a surprisingly simple trick. Finally, §5 removes inefficiencies from the basic algorithm, making it suitable for inclusion in an actual toolkit. Such a toolkit could greatly shorten the development cycle in natural language engineering.

2 Transducers and Parameters

Finite-state machines, including finite-state automata (**FSA**s) and transducers (**FST**s), are a kind of labeled directed multigraph. For ease and brevity, we explain them by example. Fig. 1a shows a probabilistic FST with input alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$, output alphabet $\Delta = \{\texttt{x}, \texttt{z}\}$, and all states final. It may be regarded as a device for generating a string pair in $\Sigma^* \times \Delta^*$ by a random walk from ⓪. Two paths exist that generate both input aabb and output xz:

$$\textcircled{0} \xrightarrow{\texttt{a:x}/.63} \textcircled{0} \xrightarrow{\texttt{a:}\epsilon/.07} \textcircled{1} \xrightarrow{\texttt{b:}\epsilon/.03} \textcircled{2} \xrightarrow{\texttt{b:z}/.4} \textcircled{2/.5}$$

$$\textcircled{0} \xrightarrow{\texttt{a:x}/.63} \textcircled{0} \xrightarrow{\texttt{a:}\epsilon/.07} \textcircled{1} \xrightarrow{\texttt{b:z}/.12} \textcircled{2} \xrightarrow{\texttt{b:}\epsilon/.1} \textcircled{2/.5}$$

Each of the paths has probability .0002646, so the probability of somehow generating the pair $(\texttt{aabb}, \texttt{xz})$ is $.0002646 + .0002646 = .0005292$.

Abstracting away from the idea of random walks, arc weights need not be probabilities. Still, define a path's weight as the product of its arc weights and the stopping weight of its final state. Thus Fig. 1a defines a weighted relation f where $f(\texttt{aabb}, \texttt{xz}) = .0005292$. This particular relation does happen to be probabilistic (see §1). It represents a joint distribution (since $\sum_{x,y} f(x,y) = 1$). Meanwhile, Fig. 1c defines a conditional one ($\forall x \sum_y f(x,y) = 1$).

This paper explains how to adjust probability distributions like that of Fig. 1a so as to model training data better. The algorithm improves an FST's numeric weights while leaving its topology fixed.

How many parameters are there to adjust in Fig. 1a? That is up to the user who built it! An FST model with few parameters is more constrained, making optimization easier. Some possibilities:

- Most simply, the algorithm can be asked to tune the 17 numbers in Fig. 1a separately, subject to the constraint that the paths retain total probability 1. A more specific version of the constraint requires the FST to remain **Markovian**: each of the 4 states must present options with total probability 1 (at state ①, 15+.7+.03.+.12=1). This preserves the random-walk interpretation and (we will show) entails no loss of generality. The 4 restrictions leave $\underline{13}$ free params.
- But perhaps Fig. 1a was actually obtained as the composition of Fig. 1b–c, effectively defining $P(input, output) = \sum_{mid} P(input, mid) \cdot P(output \mid mid)$. If Fig. 1b–c are required to remain Markovian, they have 5 and 1 degrees of freedom respectively, so now Fig. 1a has only $\underline{6}$ parameters total.[2] In general, composing machines multiplies their arc counts but only adds their parameter counts. We wish to optimize just the few underlying parameters, not independently optimize the many arc weights of the composed machine.
- Perhaps Fig. 1b was itself obtained by the probabilistic regular expression $(\texttt{a}:\texttt{p})*_\lambda(\texttt{b}:(\texttt{p} +_\mu \texttt{q}))*_\nu$ with the 3 parameters $(\lambda, \mu, \nu) = (.7, .2, .5)$. With $\rho = .1$ from footnote 2, the composed machine

[2] Why does Fig. 1c have only 1 degree of freedom? The Markovian requirement means something different in Fig. 1c, which defines a conditional relation $P(output \mid mid)$ rather than a joint one. A random walk on Fig. 1c chooses among arcs with a *given* input label. So the arcs from state ⑥ *with input* p must have total probability 1 (currently .9+.1). All other arc choices are forced by the input label and so have probability 1. The only tunable value is .1 (denote it by ρ), with $.9 = 1 - \rho$.

(Fig. 1a) has now been described with a total of just $\underline{4}$ parameters![3] Here, probabilistic union $E +_\mu F \overset{\text{def}}{=} \mu E + (1-\mu)F$ means "flip a μ-weighted coin and generate E if heads, F if tails." $E*_\lambda \overset{\text{def}}{=} (\lambda E)^*(1-\lambda)$ means "repeatedly flip an λ-weighted coin and keep repeating E as long as it comes up heads."

These 4 parameters have global effects on Fig. 1a, thanks to complex parameter tying: arcs $\textcircled{4}\xrightarrow{\text{b:p}}\textcircled{5}$, $\textcircled{5}\xrightarrow{\text{b:q}}\textcircled{5}$ in Fig. 1b get respective probabilities $(1-\lambda)\mu\nu$ and $(1-\mu)\nu$, which covary with ν and vary oppositely with μ. Each of these probabilities in turn affects multiple arcs in the composed FST of Fig. 1a.

We offer a theorem that highlights the broad applicability of these modeling techniques.[4] If $f(input, output)$ is a weighted regular relation, then the following statements are equivalent: (1) f is a joint probabilistic relation; (2) f can be computed by a Markovian FST that halts with probability 1; (3) f can be expressed as a **probabilistic regexp**, i.e., a regexp built up from atomic expressions $a : b$ (for $a \in \Sigma \cup \{\epsilon\}, b \in \Delta \cup \{\epsilon\}$) using concatenation, probabilistic union $+_p$, and probabilistic closure $*_p$.

For defining *conditional* relations, a good regexp language is unknown to us, but they can be defined in several other ways: (1) via FSTs as in Fig. 1c, (2) by compilation of weighted rewrite rules (Mohri and Sproat, 1996), (3) by compilation of decision trees (Sproat and Riley, 1996), (4) as a relation that performs contextual left-to-right replacement of input substrings by a smaller conditional relation (Gerdemann and van Noord, 1999),[5] (5) by conditionalization of a joint relation as discussed below.

A central technique is to define a joint relation as a **noisy-channel model**, by composing a joint relation with a cascade of one or more conditional relations as in Fig. 1 (Pereira and Riley, 1997; Knight and Graehl, 1998). The general form is illustrated by $P(v, z) \overset{\text{def}}{=} \sum_{w,x,y} P(v|w)P(w, x)P(y|x)P(z|y)$, implemented by composing 4 machines.[6,7]

There are also procedures for defining weighted FSTs that are not probabilistic (Berstel and Reutenauer, 1988). Arbitrary weights such as 2.7 may be assigned to arcs or sprinkled through a regexp (to be compiled into $\xrightarrow{\epsilon:\epsilon/2.7}$ arcs). A more subtle example is weighted FSAs that approximate PCFGs (Nederhof, 2000; Mohri and Nederhof, 2001), or to extend the idea, weighted FSTs that approximate joint or conditional *synchronous* PCFGs built for translation. These are parameterized by the PCFG's parameters, but add or remove strings of the PCFG to leave an improper probability distribution.

Fortunately for those techniques, an FST with positive arc weights can be **normalized** to make it jointly or conditionally probabilistic:

- An easy approach is to normalize the options at each state to make the FST Markovian. Unfortunately, the result may differ for equivalent FSTs that express the same weighted relation. Undesirable consequences of this fact have been termed "label bias" (Lafferty et al., 2001). Also, in the conditional case such **per-state normalization** is only correct if all states accept all input suffixes (since "dead ends" leak probability mass)[8] and all input labels fall in Σ.
- A better-founded approach is **global normalization**, which simply divides each $f(x, y)$ by $\sum_{x',y'} f(x', y')$ (joint case) or by $\sum_{y'} f(x, y')$ (conditional case). To implement the joint case, just divide stopping weights by the total weight of all paths (which §4 shows how to find), provided this is finite. In the conditional case, let g be a copy of f with the output labels removed, so that $g(x)$ finds the desired divisor; determinize g if possible (but this fails for some weighted FSAs), replace all weights with their reciprocals, and compose the result with f.[9]

[3] Conceptually, the parameters represent the probabilities of reading another a (λ); reading another b (ν); transducing b to p rather than q (μ); starting to transduce p to ϵ rather than x (ρ).

[4] To prove (1)⇒(3), express f as an FST and apply the well-known Kleene-Schützenberger construction (Berstel and Reutenauer, 1988), taking care to write each regexp in the construction as a constant times a probabilistic regexp. A full proof is straightforward, as are proofs of (3)⇒(2), (2)⇒(1).

[5] In (4), the randomness is in the smaller relation's choice of how to replace a match. One can also get randomness through the choice of matches, ignoring match possibilities by randomly deleting markers in Gerdemann and van Noord's construction.

[6] $P(w, x)$ defines the source model, and is often an "identity FST" that requires $w = x$, really just an FSA.

[7] We propose also using n-tape automata to generalize to "branching noisy channels" (a case of dendroid distributions). In $\sum_{w,x} P(v|w)P(v'|w)P(w, x)P(y|x)$, the true transcription w can be triply constrained by observing speech y and *two* errorful transcriptions v, v', which independently depend on w.

[8] A corresponding problem exists in the joint case, but may be easily avoided there by first pruning non-coaccessible states.

[9] It suffices to make g unambiguous (one accepting path per string), a weaker condition than determinism. When this is not possible (as in the inverse of Fig. 1b, whose conditionaliza-

Normalization is particularly important because it enables the use of **log-linear** (maximum-entropy) parameterizations. Here one defines each arc weight, coin weight, or regexp weight in terms of meaningful **features** associated by hand with that arc, coin, etc. Each feature has a **strength** $\in \mathbb{R}_{>0}$, and a weight is computed as the product of the strengths of its features.[10] It is now the strengths that are the learnable parameters. This allows meaningful parameter tying: if certain arcs such as $\xrightarrow{\texttt{u:i}}$, $\xrightarrow{\texttt{o:e}}$, and $\xrightarrow{a\texttt{:æ}}$ share a contextual "vowel-fronting" feature, then their weights rise and fall together with the strength of that feature. The resulting machine must be normalized, either per-state or globally, to obtain a joint or a conditional distribution as desired. Such approaches have been tried recently in restricted cases (McCallum et al., 2000; Eisner, 2001b; Lafferty et al., 2001).

Normalization may be postponed and applied instead to the result of combining the FST with other FSTs by composition, union, concatenation, etc. A simple example is a probabilistic FSA defined by normalizing the intersection of other probabilistic FSAs $f_1, f_2, \ldots$. (This is in fact a log-linear model in which the component FSAs define the features: string x has $\log f_i(x)$ occurrences of feature i.)

In short, weighted finite-state operators provide a language for specifying a wide variety of parameterized statistical models. Let us turn to their training.

3 Estimation in Parameterized FSTs

We are primarily concerned with the following training paradigm, novel in its generality. Let $f_\theta : \Sigma^* \times \Delta^* \to \mathbb{R}_{\geq 0}$ be a joint probabilistic relation that is computed by a weighted FST. The FST was built by some recipe that used the **parameter vector** θ. Changing θ may require us to rebuild the FST to get updated weights; this can involve composition, regexp compilation, multiplication of feature strengths, etc. (Lazy algorithms that compute arcs and states of f_θ on demand (Mohri et al., 1998) can pay off here, since only part of f_θ may be needed subsequently.)

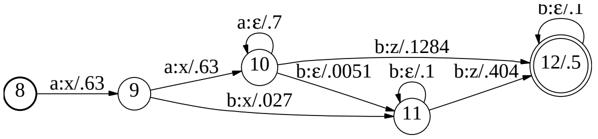

Figure 2: The joint model of Fig. 1a constrained to generate only input $\in \texttt{a}(\texttt{a} + \texttt{b})^*$ and output $= \texttt{xxz}$.

As training data we are given a set of observed $(input, output)$ pairs, (x_i, y_i). These are assumed to be independent random samples from a joint distribution of the form $f_{\hat{\theta}}(x, y)$; the goal is to recover the true $\hat{\theta}$. Samples need not be fully observed (partly supervised training): thus $x_i \subseteq \Sigma^*, y_i \subseteq \Delta^*$ may be given as regular sets in which input and output were observed to fall. For example, in ordinary HMM training, $x_i = \Sigma^*$ and represents a completely hidden state sequence (cf. Ristad (1998), who allows any regular set), while y_i is a single string representing a completely observed emission sequence.[11]

What to optimize? **Maximum-likelihood estimation** guesses $\hat{\theta}$ to be the θ maximizing $\prod_i f_\theta(x_i, y_i)$. **Maximum-posterior estimation** tries to maximize $P(\theta) \cdot \prod_i f_\theta(x_i, y_i)$ where $P(\theta)$ is a prior probability. In a log-linear parameterization, for example, a prior that penalizes feature strengths far from 1 can be used to do feature selection and avoid overfitting (Chen and Rosenfeld, 1999).

The EM algorithm (Dempster et al., 1977) can maximize these functions. Roughly, the **E step** guesses hidden information: if (x_i, y_i) was generated from the current f_θ, which FST paths stand a chance of having been the path used? (Guessing the path also guesses the exact input and output.) The **M step** updates θ to make those paths more likely. EM alternates these steps and converges to a local optimum. The M step's form depends on the parameterization and the E step serves the M step's needs.

Let f_θ be Fig. 1a and suppose $(x_i, y_i) = (\texttt{a}(\texttt{a} + \texttt{b})^*, \texttt{xxz})$. During the E step, we restrict to paths compatible with this observation by computing $x_i \circ f_\theta \circ y_i$, shown in Fig. 2. To find each path's posterior probability *given the observation* (x_i, y_i), just conditionalize: divide its raw probability by the total probability (≈ 0.1003) of all paths in Fig. 2.

tion cannot be realized by any weighted FST), one can sometimes succeed by *first* intersecting g with a smaller regular set in which the input being considered is known to fall. In the extreme, if each input string is fully observed (not the case if the input is bound by composition to the output of a one-to-many FST), one can succeed by restricting g to each input string in turn; this amounts to manually dividing $f(x, y)$ by $g(x)$.

[10] Traditionally log(strength) values are called weights, but this paper uses "weight" to mean something else.

[11] To implement an HMM by an FST, compose a probabilistic FSA that generates a state sequence of the HMM with a conditional FST that transduces HMM states to emitted symbols.

But that is not the full E step. The M step uses not individual path probabilities (Fig. 2 has infinitely many) but expected counts derived from the paths. Crucially, §4 will show how the E step can accumulate these counts effortlessly. We first explain their use by the M step, repeating the presentation of §2:

- If the parameters are the 17 weights in Fig. 1a, the M step reestimates the probabilities of the arcs from each state to be proportional to the *expected number of traversals* of each arc (normalizing at each state to make the FST Markovian). So the E step must count traversals. This requires mapping Fig. 2 back onto Fig. 1a: to traverse either $\textcircled{8}\xrightarrow{\text{a:x}}\textcircled{9}$ or $\textcircled{9}\xrightarrow{\text{a:x}}\textcircled{10}$ in Fig. 2 is "really" to traverse $\textcircled{0}\xrightarrow{\text{a:x}}\textcircled{0}$ in Fig. 1a.
- If Fig. 1a was built by composition, the M step is similar but needs the expected traversals of the arcs in Fig. 1b–c. This requires further unwinding of Fig. 1a's $\textcircled{0}\xrightarrow{\text{a:x}}\textcircled{0}$: to traverse that arc is "really" to traverse Fig. 1b's $\textcircled{4}\xrightarrow{\text{a:p}}\textcircled{4}$ *and* Fig. 1c's $\textcircled{6}\xrightarrow{\text{p:x}}\textcircled{6}$.
- If Fig. 1b was defined by the regexp given earlier, traversing $\textcircled{4}\xrightarrow{\text{a:p}}\textcircled{4}$ is in turn "really" just evidence that the λ-coin came up heads. To learn the weights λ, ν, μ, ρ, count *expected heads/tails* for each coin.
- If arc probabilities (or even λ, ν, μ, ρ) have log-linear parameterization, then the E step must compute $c = \sum_i ec_f(x_i, y_i)$, where $ec(x, y)$ denotes the *expected vector of total feature counts* along a random path in f_θ whose $(input, output)$ matches (x, y). The M step then treats c as fixed, *observed* data and adjusts θ until the *predicted* vector of total feature counts equals c, using Improved Iterative Scaling (Della Pietra et al., 1997; Chen and Rosenfeld, 1999).[12] For globally normalized, joint models, the predicted vector is $ec_f(\Sigma^*, \Delta^*)$. If the log-linear probabilities are conditioned on the state and/or the input, the predicted vector is harder to describe (though usually much easier to compute).[13]

It is also possible to use this EM approach for **discriminative training**, where we wish to maximize $\prod_i P(y_i \mid x_i)$ and $f_\theta(x, y)$ is a conditional FST that defines $P(y \mid x)$. The trick is to instead train a joint model $g \circ f_\theta$, where $g(x_i)$ defines $P(x_i)$, thereby maximizing $\prod_i P(x_i) \cdot P(y_i \mid x_i)$. (Of course, the method of this paper can train such compositions.) If $x_1, \ldots x_n$ are fully observed, just define each $g(x_i) = 1/n$. But by choosing a more general model of g, we can also handle incompletely observed x_i: training $g \circ f_\theta$ then forces g and f_θ to cooperatively reconstruct a distribution over the possible inputs and do discriminative training of f_θ given those inputs. (Any parameters of g may be either frozen before training or optimized along with the parameters of f_θ.) A final possibility is that each x_i is defined by a *probabilistic* FSA that already supplies a distribution over the inputs; then we consider $x_i \circ f_\theta \circ y_i$ directly, just as in the joint model.

Finally, note that EM is not all-purpose. It only maximizes probabilistic objective functions, and even there it is not necessarily as fast as (say) conjugate gradient. For this reason, we will also show below how to compute the gradient of $f_\theta(x_i, y_i)$ with respect to θ, for an arbitrary parameterized FST f_θ. We remark without elaboration that this can help optimize task-related objective functions, such as $\sum_i \sum_y (P(x_i, y)^\alpha / \sum_{y'} P(x_i, y')^\alpha) \cdot error(y, y_i)$.

4 The E Step: Expectation Semirings

It remains to devise appropriate E steps, which looks rather daunting. Each path in Fig. 2 weaves together parameters from other machines, which we must untangle and tally. In the 4-coin parameterization, path $\textcircled{8}\xrightarrow{a:x}\textcircled{9}\xrightarrow{a:x}\textcircled{10}\xrightarrow{a:\epsilon}\textcircled{10}\xrightarrow{a:\epsilon}\textcircled{10}\xrightarrow{b:z}\textcircled{12}$ must yield up a vector $\langle H_\lambda, T_\lambda, H_\mu, T_\mu, H_\nu, T_\nu, H_\rho, T_\rho \rangle$ that counts observed heads and tails of the 4 coins. This nontrivially works out to $\langle 4, 1, 0, 1, 1, 1, 1, 2 \rangle$. For other parameterizations, the path must instead yield a vector of arc traversal counts or feature counts.

Computing a count vector for one path is hard enough, but it is the E step's job to find the expected value of this vector—an average over the infinitely

[12] IIS is itself iterative; to avoid nested loops, run only one iteration at each M step, giving a GEM algorithm (Riezler, 1999). Alternatively, discard EM and use gradient-based optimization.

[13] For per-state conditional normalization, let $D_{j,a}$ be the set of arcs from state j with input symbol $a \in \Sigma$; their weights are normalized to sum to 1. Besides computing c, the E step must count the expected number $d_{j,a}$ of traversals of arcs in each $D_{j,a}$. Then the predicted vector given θ is $\sum_{j,a} d_{j,a} \cdot$ (expected feature counts on a randomly chosen arc in $D_{j,a}$). Per-state joint normalization (Eisner, 2001b, §8.2) is similar but drops the dependence on a. The difficult case is global conditional normalization. It arises, for example, when training a joint model of the form $f_\theta = \cdots (g_\theta \circ h_\theta) \cdots$, where h_θ is a conditional log-linear model of $P(v \mid u)$ for $u \in \Sigma'^*, v \in \Delta'^*$. Then the predicted count vector contributed by h is $\sum_i \sum_{u \in \Sigma'^*} P(u \mid x_i, y_i) \cdot ec_h(u, \Delta'^*)$. The term $\sum_i P(u \mid x_i, y_i)$ computes the expected count of each $u \in \Sigma'^*$. It may be found by a variant of §4 in which path values are regular expressions over Σ'^*.

many paths π through Fig. 2 in proportion to their posterior probabilities $P(\pi \mid x_i, y_i)$. The results for all (x_i, y_i) are summed and passed to the M step.

Abstractly, let us say that each path π has not only a probability $P(\pi) \in [0, 1]$ but also a **value** $\mathrm{val}(\pi)$ in a vector space V, which counts the arcs, features, or coin flips encountered along path π. The value of a path is the *sum* of the values assigned to its arcs. The E step must return the **expected value** of the unknown path that generated (x_i, y_i). For example, if every arc had value 1, then expected value would be expected path length. Letting Π denote the set of paths in $x_i \circ f_\theta \circ y_i$ (Fig. 2), the expected value is[14]

$$\mathbf{E}[\mathrm{val}(\pi) \mid x_i, y_i] = \frac{\sum_{\pi \in \Pi} P(\pi)\,\mathrm{val}(\pi)}{\sum_{\pi \in \Pi} P(\pi)} \qquad (1)$$

The denominator of equation (1) is the total probability of all accepting paths in $x_i \circ f \circ y_i$. But while computing this, we will also compute the numerator. The idea is to augment the weight data structure with expectation information, so each weight records a probability *and* a vector counting the parameters that contributed to that probability. We will enforce an **invariant**: the weight of *any* pathset Π must be $(\sum_{\pi \in \Pi} P(\pi), \sum_{\pi \in \Pi} P(\pi)\,\mathrm{val}(\pi)) \in \mathbb{R}_{\geq 0} \times V$, from which (1) is trivial to compute.

Berstel and Reutenauer (1988) give a sufficiently general finite-state framework to allow this: weights may fall in any set K (instead of $\mathbb{R}$). Multiplication and addition are replaced by binary operations $\otimes$ and $\oplus$ on K. Thus $\otimes$ is used to combine arc weights into a path weight and $\oplus$ is used to combine the weights of alternative paths. To sum over infinite sets of cyclic paths we also need a closure operation *, interpreted as $k^* = \bigoplus_{i=0}^{\infty} k^i$. The usual finite-state algorithms work if $(K, \oplus, \otimes, ^*)$ has the structure of a **closed semiring**.[15]

Ordinary probabilities fall in the semiring $(\mathbb{R}_{\geq 0}, +, \times, ^*)$.[16] Our novel weights fall in a novel ***V*-expectation semiring**, $(\mathbb{R}_{\geq 0} \times V, \oplus, \otimes, ^*)$:

$$(p_1, v_1) \otimes (p_2, v_2) \overset{\text{def}}{=} (p_1 p_2, p_1 v_2 + v_1 p_2) \qquad (2)$$

$$(p_1, v_1) \oplus (p_2, v_2) \overset{\text{def}}{=} (p_1 + p_2, v_1 + v_2) \qquad (3)$$

$$\text{if } p^* \text{ defined, } (p, v)^* \overset{\text{def}}{=} (p^*, p^* v p^*) \qquad (4)$$

If an arc has probability p and value v, we give it the weight (p, pv), so that our invariant (see above) holds if Π consists of a single length-0 or length-1 path. The above definitions are designed to preserve our invariant as we build up larger paths and pathsets. $\otimes$ lets us concatenate (e.g.) simple paths π_1, π_2 to get a longer path π with $P(\pi) = P(\pi_1)P(\pi_2)$ and $\mathrm{val}(\pi) = \mathrm{val}(\pi_1) + \mathrm{val}(\pi_2)$. The definition of $\otimes$ guarantees that path π's weight will be $(P(\pi), P(\pi) \cdot \mathrm{val}(\pi))$. $\oplus$ lets us take the union of two disjoint pathsets, and * computes infinite unions.

To compute (1) now, we only need the total weight t_i of accepting paths in $x_i \circ f \circ y_i$ (Fig. 2). This can be computed with finite-state methods: the machine $(\epsilon \times x_i) \circ f \circ (y_i \times \epsilon)$ is a version that replaces all input:output labels with $\epsilon : \epsilon$, so it maps (ϵ, ϵ) to the same total weight t_i. Minimizing it yields a one-state FST from which t_i can be read directly!

The other "magical" property of the expectation semiring is that it automatically keeps track of the tangled parameter counts. For instance, recall that traversing $⓪ \xrightarrow{\text{a:x}} ⓪$ should have the same effect as traversing *both* the underlying arcs $④ \xrightarrow{\text{a:p}} ④$ and $⑥ \xrightarrow{\text{p:x}} ⑥$. And indeed, if the underlying arcs have values v_1 and v_2, then the composed arc $⓪ \xrightarrow{\text{a:x}} ⓪$ gets weight $(p_1, p_1 v_1) \otimes (p_2, p_2 v_2) = (p_1 p_2, p_1 p_2 (v_1 + v_2))$, just as if it had value $v_1 + v_2$.

Some concrete examples of values may be useful:

- To count traversals of the arcs of Figs. 1b–c, number these arcs and let arc ℓ have value e_ℓ, the ℓ^{th} basis vector. Then the ℓ^{th} element of $\mathrm{val}(\pi)$ counts the appearances of arc ℓ in path π, or underlying path π.
- A regexp of form $E +_\mu F = \mu E + (1-\mu) F$ should be weighted as $(\mu, \mu e_k) E + (1 - \mu, (1 - \mu) e_{k+1}) F$ in the new semiring. Then elements k and $k + 1$ of $\mathrm{val}(\pi)$ count the heads and tails of the μ-coin.
- For a global log-linear parameterization, an arc's value is a vector specifying the arc's features. Then $\mathrm{val}(\pi)$ counts all the features encountered along π.

Really we are manipulating weighted relations, not FSTs. We may combine FSTs, or determinize

[14] Formal derivation of (1): $\sum_\pi P(\pi \mid x_i, y_i)\,\mathrm{val}(\pi) = (\sum_\pi P(\pi, x_i, y_i)\,\mathrm{val}(\pi))/P(x_i, y_i) = (\sum_\pi P(x_i, y_i \mid \pi) P(\pi)\,\mathrm{val}(\pi)) / \sum_\pi P(x_i, y_i \mid \pi) P(\pi)$; now observe that $P(x_i, y_i \mid \pi) = 1$ or 0 according to whether $\pi \in \Pi$.

[15] That is: $(K, \otimes)$ is a monoid (i.e., $\otimes : K \times K \to K$ is associative) with identity $\underline{1}$. $(K, \oplus)$ is a *commutative* monoid with identity $\underline{0}$. $\otimes$ distributes over $\oplus$ from both sides, $\underline{0} \otimes k = k \otimes \underline{0} = \underline{0}$, and $k^* = \underline{1} \oplus k \otimes k^* = \underline{1} \oplus k^* \otimes k$. For finite-state composition, commutativity of $\otimes$ is needed as well.

[16] The closure operation is defined for $p \in [0, 1)$ as $p^* = 1/(1 - p)$, so cycles with weights in $[0, 1)$ are allowed.

or minimize them, with any variant of the semiring-weighted algorithms.[17] As long as the resulting FST computes the right weighted relation, the arrangement of its states, arcs, and labels is unimportant.

The same semiring may be used to compute gradients. We would like to find $f_\theta(x_i, y_i)$ and its gradient with respect to θ, where f_θ is real-valued but need not be probabilistic. Whatever procedures are used to evaluate $f_\theta(x_i, y_i)$ exactly or approximately—for example, FST operations to compile f_θ followed by minimization of $(\epsilon \times x_i) \circ f_\theta \circ (y_i \times \epsilon)$—can simply be applied over the expectation semiring, replacing each weight p by $(p, \nabla p)$ and replacing the usual arithmetic operations with $\oplus$, $\otimes$, etc.[18] (2)–(4) preserve the gradient ((2) is the derivative product rule), so this computation yields $(f_\theta(x_i, y_i), \nabla f_\theta(x_i, y_i))$.

5 Removing Inefficiencies

Now for some important remarks on efficiency:

- Computing t_i is an instance of the well-known **algebraic path problem** (Lehmann, 1977; Tarjan, 1981a). Let $T_i = x_i \circ f \circ y_i$. Then t_i is the total semiring weight w_{0n} of paths in T_i from initial state 0 to final state n (assumed WLOG to be unique and unweighted). It is wasteful to compute t_i as suggested earlier, by minimizing $(\epsilon \times x_i) \circ f \circ (y_i \times \epsilon)$, since then the real work is done by an ϵ-closure step (Mohri, 2002) that implements the **all-pairs** version of algebraic path, whereas all we need is the **single-source** version. If n and m are the number of states and edges,[19] then both problems are $O(n^3)$ in the worst case, but the single-source version can be solved in essentially $O(m)$ time for acyclic graphs and other reducible flow graphs (Tarjan, 1981b). For a general graph T_i, Tarjan (1981b) shows how to partition into "hard" subgraphs that localize the cyclicity or irreducibility, then run the $O(n^3)$ algorithm on each subgraph (thereby reducing n to as little as 1), and recombine the results. The overhead of partitioning and recombining is essentially only $O(m)$.
- For speeding up the $O(n^3)$ problem on subgraphs, one can use an approximate relaxation technique (Mohri, 2002). Efficient hardware implementation is also possible via chip-level parallelism (Rote, 1985).
- In many cases of interest, T_i is an acyclic graph.[20] Then Tarjan's method computes w_{0j} for each j in topologically sorted order, thereby finding t_i in a linear number of $\oplus$ and $\otimes$ operations. For HMMs (footnote 11), T_i is the familiar trellis, and we would like this computation of t_i to reduce to the forward-backward algorithm (Baum, 1972). But notice that it has no backward pass. In place of pushing cumulative probabilities backward to the arcs, it pushes cumulative arcs (more generally, values in V) forward to the probabilities. This is slower because our $\oplus$ and $\otimes$ are vector operations, and the vectors rapidly lose sparsity as they are added together. We therefore reintroduce a backward pass that lets us avoid $\oplus$ and $\otimes$ when computing t_i (so they are needed only to construct T_i). This speedup also works for cyclic graphs and for any V. Write w_{jk} as (p_{jk}, v_{jk}), and let $w^1_{jk} = (p^1_{jk}, v^1_{jk})$ denote the weight of the edge from j to k.[19] Then it can be shown that $w_{0n} = (p_{0n}, \sum_{j,k} p_{0j} v^1_{jk} p_{kn})$. The forward and backward probabilities, p_{0j} and p_{kn}, can be computed using single-source algebraic path for the simpler semiring $(\mathbb{R}, +, \times, {}^*)$—or equivalently, by solving a sparse linear system of equations over $\mathbb{R}$, a much-studied problem at $O(n)$ space, $O(nm)$ time, and faster approximations (Greenbaum, 1997).
- A Viterbi variant of the expectation semiring exists: replace (3) with $\text{if}(p_1 > p_2, (p_1, v_1), (p_2, v_2))$. Here, the forward and backward probabilities can be computed in time only $O(m + n \log n)$ (Fredman and Tarjan, 1987). k-best variants are also possible.

6 Discussion

We have exhibited a training algorithm for parameterized finite-state machines. Some specific consequences that we believe to be novel are (1) an EM algorithm for FSTs with cycles and epsilons; (2) training algorithms for HMMs and weighted contextual edit distance that work on incomplete data; (3) end-to-end training of noisy channel cascades, so that it is not necessary to have separate training data for each machine in the cascade (cf. Knight and Graehl,

[17] Eisner (submitted) develops fast minimization algorithms that work for the real and V-expectation semirings.

[18] Division and subtraction are also possible: $-(p, v) = (-p, -v)$ and $(p, v)^{-1} = (p^{-1}, -p^{-1}vp^{-1})$. Division is commonly used in defining f_θ (for normalization).

[19] Multiple edges from j to k are summed into a single edge.

[20] If x_i and y_i are acyclic (e.g., fully observed strings), and f (or rather its FST) has no $\epsilon : \epsilon$ cycles, then composition will "unroll" f into an acyclic machine. If only x_i is acyclic, then the composition is still acyclic if domain(f) has no ϵ cycles.

1998), although such data could also be used; (4) training of branching noisy channels (footnote 7); (5) discriminative training with incomplete data; (6) training of conditional MEMMs (McCallum et al., 2000) and conditional random fields (Lafferty et al., 2001) on unbounded sequences.

We are particularly interested in the potential for quickly building statistical models that incorporate linguistic and engineering insights. Many models of interest can be constructed in our paradigm, without having to write new code. Bringing diverse models into the same declarative framework also allows one to apply new optimization methods, objective functions, and finite-state algorithms to all of them.

To avoid local maxima, one might try deterministic annealing (Rao and Rose, 2001), or randomized methods, or place a prior on θ. Another extension is to adjust the machine *topology*, say by model merging (Stolcke and Omohundro, 1994). Such techniques build on our parameter estimation method.

The key algorithmic ideas of this paper extend from forward-backward-style to inside-outside-style methods. For example, it should be possible to do end-to-end training of a weighted relation defined by an interestingly parameterized synchronous CFG composed with tree transducers and then FSTs.

References

L. E. Baum. 1972. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, 3.

Jean Berstel and Christophe Reutenauer. 1988. *Rational Series and their Languages*. Springer-Verlag.

Stanley F. Chen and Ronald Rosenfeld. 1999. A Gaussian prior for smoothing maximum entropy models. Technical Report CMU-CS-99-108, Carnegie Mellon.

S. Della Pietra, V. Della Pietra, and J. Lafferty. 1997. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4).

A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statist. Soc. Ser. B*, 39(1):1–38.

Jason Eisner. 2001a. Expectation semirings: Flexible EM for finite-state transducers. In G. van Noord, ed., *Proc. of the ESSLLI Workshop on Finite-State Methods in Natural Language Processing*. Extended abstract.

Jason Eisner. 2001b. *Smoothing a Probabilistic Lexicon via Syntactic Transformations*. Ph.D. thesis, University of Pennsylvania.

D. Gerdemann and G. van Noord. 1999. Transducers from rewrite rules with backreferences. *Proc. of EACL*.

Anne Greenbaum. 1997. *Iterative Methods for Solving Linear Systems*. Soc. for Industrial and Applied Math.

Kevin Knight and Yaser Al-Onaizan. 1998. Translation with finite-state devices. In *Proc. of AMTA*.

Kevin Knight and Jonathan Graehl. 1998. Machine transliteration. *Computational Linguistics*, 24(4).

J. Lafferty, A. McCallum, and F. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proc. of ICML*.

D. J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76.

A. McCallum, D. Freitag, and F. Pereira. 2000. Maximum entropy Markov models for information extraction and segmentation. *Proc. of ICML*, 591–598.

M. Mohri and M.-J. Nederhof. 2001. Regular approximation of context-free grammars through transformation. In J.-C. Junqua and G. van Noord, eds., *Robustness in Language and Speech Technology*. Kluwer.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proc. of ACL*.

M. Mohri, F. Pereira, and M. Riley. 1998. A rational design for a weighted finite-state transducer library. *Lecture Notes in Computer Science*, 1436.

M. Mohri. 2002. Generic epsilon-removal and input epsilon-normalization algorithms for weighted transducers. *Int. J. of Foundations of Comp. Sci.*, 1(13).

Mark-Jan Nederhof. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1).

Fernando C. N. Pereira and Michael Riley. 1997. Speech recognition by composition of weighted finite automata. In E. Roche and Y. Schabes, eds., *Finite-State Language Processing*. MIT Press, Cambridge, MA.

A. Rao and K. Rose. 2001 Deterministically annealed design of hidden Markov movel speech recognizers. In *IEEE Trans. on Speech and Audio Processing*, 9(2).

Stefan Riezler. 1999. *Probabilistic Constraint Logic Programming*. Ph.D. thesis, Universität Tübingen.

E. Ristad and P. Yianilos. 1996. Learning string edit distance. Tech. Report CS-TR-532-96, Princeton.

E. Ristad. 1998. Hidden Markov models with finite state supervision. In A. Kornai, ed., *Extended Finite State Models of Language*. Cambridge University Press.

Emmanuel Roche and Yves Schabes, editors. 1997. *Finite-State Language Processing*. MIT Press.

Günter Rote. 1985. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34(3):191–219.

Richard Sproat and Michael Riley. 1996. Compilation of weighted finite-state transducers from decision trees. In *Proceedings of the 34th Annual Meeting of the ACL*.

Andreas Stolcke and Stephen M. Omohundro. 1994. Best-first model merging for hidden Markov model induction. Tech. Report ICSI TR-94-003, Berkeley, CA.

Robert Endre Tarjan. 1981a. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July.

Robert Endre Tarjan. 1981b. Fast algorithms for solving path problems. *J. of the ACM*, 28(3):594–614, July.

G. van Noord and D. Gerdemann. 2001. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Automata Implementation*, no. 22 in Springer Lecture Notes in CS.