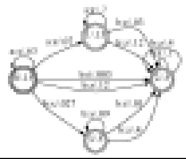
Parameterized Finite-State Machines and their Training

Jason Eisner
Johns Hopkins University

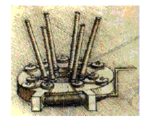
October 16, 2002 — AT&T Speech Days



Outline – The Vision Slide!

1. Finite-state machines as a shared modeling language.

2. The training gizmo (an algorithm).



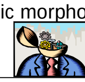
Should use out of-the-box finite-state gizmos to build and train most of our current models. Easier, faster, better, & enables fancier models.

Training Probabilistic FSMs

- State of the world – surprising:
  - Training for HMMs, alignment, many variants
  - But no basic training algorithm for **all** FSAs
  - Fancy toolkits for building them, but no learning
- New algorithm:
  - Training for FSAs, FSTs, ... (collectively FSMs)
  - Supervised, unsupervised, incompletely supervised ...
  - Train components separately or all at once
  - Epsilon-cycles OK
  - Complicated parameterizations OK

"If you build it, it will train"

Currently Two Finite-State Camps

	Vanilla FSTs	Probabilistic FSTs
What they represent	Functions on strings. Or nondeterministic functions (relations).	Prob. distributions p(x,y) or p(y|x).
How they're currently used	Encode expert knowledge about Arabic morphology, etc. 	Noisy channel models p(x)p(y|x)p(z|y)... (much more limited)
How they're currently built	Fancy regular expressions (or sometimes TBL)	Build parts by hand For each part, get arc weights somehow Then combine parts (much more limited)

Knight & Graehl 1997 - transliteration

Current Limitation

- Big FSM must be made of separately trainable parts.

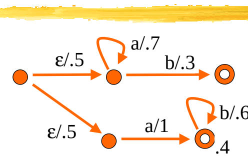
p(English text)
o
p(English text → English phonemes)
o
p(English phonemes → Japanese phonemes)
o
p(Japanese phonemes → Japanese text)

Need explicit training data for this part (smaller loanword corpus).
A pity – would like to use guesses.

Topology must be simple enough to train by current methods.
A pity – would like to get some of that expert knowledge in here!

Topology: sensitive to syllable struct?
Parameterization: /t/ and /d/ are similar phonemes ... parameter tying?

Probabilistic FSA

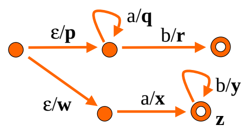


Example: ab is accepted along 2 paths
p(ab) = (.5 .7 .3) + (.5 .6 .4) = .225

Regexp: $(a^{*.7}\ b)\ +_{.5}\ (ab^{*.6})$

Theorem: Any probabilistic FSM has a regexp like this.

Weights Need Not be Reals

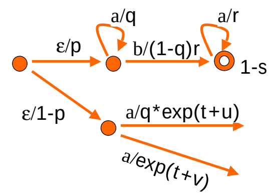


Example: ab is accepted along 2 paths
weight(ab) = (**p**⊗**q**⊗**r**) ⊕ (**w**⊗**x**⊗**y**⊗**z**)

If ⊗ ⊕ * satisfy "semiring" axioms, the finite-state constructions continue to work correctly.

Goal: Parameterized FSMs

- Parameterized FSM:
  - An FSM whose arc probabilities depend on parameters: they are formulas.

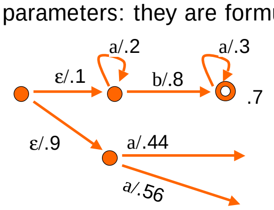


Expert first: Construct the FSM (topology & parameterization).

Automatic takes over: Given training data, find parameter values that optimize arc probs.

Goal: Parameterized FSMs

- Parameterized FSM:
  - An FSM whose arc probabilities depend on parameters: they are formulas.



Expert first: Construct the FSM (topology & parameterization).

Automatic takes over: Given training data, find parameter values that optimize arc probs.

Knight & Graehl 1997 - transliteration

Goal: Parameterized FSMs

- FSM whose arc probabilities are formulas.

p(English text)
o
p(English text → English phonemes)
o
p(English phonemes → Japanese phonemes)
o
p(Japanese phonemes → Japanese text)

"Would like to get some of that expert knowledge in here"

Use probabilistic regexps like $(a^{*.7}\ b)\ +_{.5}\ (ab^{*.6})$ …

If the probabilities are variables $(a^{*x}\ b)\ +_{y}\ (ab^{*z})$ … then arc weights of the compiled machine are nasty formulas.
(Especially after minimization!)

Knight & Graehl 1997 - transliteration

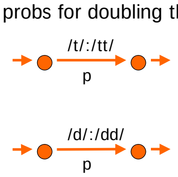
Goal: Parameterized FSMs

- An FSM whose arc probabilities are formulas.

p(English text)
o
p(English text → English phonemes)
o
p(English phonemes → Japanese phonemes)
o
p(Japanese phonemes → Japanese text)

"/t/ and /d/ are similar …"

Tied probs for doubling them:



Knight & Graehl 1997 - transliteration

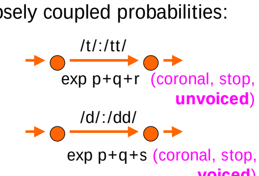
Goal: Parameterized FSMs

- An FSM whose arc probabilities are formulas.

p(English text)
o
p(English text → English phonemes)
o
p(English phonemes → Japanese phonemes)
o
p(Japanese phonemes → Japanese text)

"/t/ and /d/ are similar …"

Loosely coupled probabilities:



(with normalization)

Outline of this talk

1. What can you build with parameterized FSMs?
2. How do you train them?

Finite-State Operations

- Projection GIVES YOU marginal distribution

domain(p(x,y)) = p(x)

range(p(x,y)) = p(y)

a : ~~b~~ / 0.3   ~~a~~ : b / 0.3

Finite-State Operations

- Probabilistic union GIVES YOU mixture model

$p(x) +_{0.3} q(x)$ = 0.3 p(x) + 0.7 q(x)

0.3 p(x)

0.7 q(x)

Finite-State Operations

- Probabilistic union GIVES YOU mixture model

$p(x) +_{\alpha} q(x)$ = α p(x) + (1- α)q(x)

α p(x)

1-α q(x)

Learn the mixture parameter α!

Finite-State Operations

- Composition GIVES YOU chain rule

p(x|y) o p(y|z) = p(x|z)

p(x|y) o p(y|z) o z = p(x,z)

- The most popular statistical FSM operation
- Cross-product construction

Finite-State Operations

- Concatenation, probabilistic closure HANDLE unsegmented text

p(x) q(x) $p(x)^{*}_{0.3}$

p(x) → q(x)

p(x) 0.3 0.7

- Just glue together machines for the different segments, and let them figure out how to align with the text

Finite-State Operations

- Directed replacement MODELS noise or postprocessing

p(x,y) o D = p(x, noisy y)

noise model defined by dir. replacement

- Resulting machine compensates for noise or postprocessing

Finite-State Operations

- Intersection GIVES YOU product models
  - e.g., exponential / maxent, perceptron, Naïve Bayes, …
  - Need a normalization op too – computes $\sum_x f(x)$ "pathsum" or "partition function"

p(x) & q(x) = p(x)*q(x)

p(A(x)|y) & p(B(x)|y) & p(y) $\propto$ $p_{NB}(y \mid x)$

- Cross-product construction (like composition)

Finite-State Operations

- Conditionalization (new operation)

condit(p(x,y)) = p(y | x)

- Resulting machine can be composed with other distributions: p(y | x) * q(x)
- Construction:
  reciprocal(determinize(domain(p(x,y)))) o p(x,y)

  not possible for all weighted FSAs

Other Useful Finite-State Constructions

- Complete graphs YIELD n-gram models
- Other graphs YIELD fancy language models (skips, caching, etc.)
- Compilation from other formalism → FSM:
  - Wordlist (cf. trie), pronunciation dictionary ...
  - Speech hypothesis lattice
  - Decision tree (Sproat & Riley)
  - Weighted rewrite rules (Mohri & Sproat)
  - TBL or probabilistic TBL (Roche & Schabes)
  - PCFG (approximation!) (e.g., Mohri & Nederhof)
  - Optimality theory grammars (e.g., Eisner)
  - Logical description of set (Vaillette; Klarlund)

Regular Expression Calculus as a Modelling Language

Programming Languages	The Finite-State Case
Function	Function on strings, or probability distrib.
Source code	Regular expression (can be probabilistic)
Object code	Finite state machine
Compiler	Regexp compiler
Optimization of object code	Determinization, minimization, pruning

Regular Expression Calculus as a Modelling Language

Many features you wish other languages had!

Programming Languages	The Finite-State Case
Function composition	Machine composition
Nondeterminism	Nondeterminism
Parallelism	Compose FSA with FST
Function inversion (cf. Prolog)	Function inversion
Higher-order functions	Transform object code (apply operators to it)

Regular Expression Calculus as a Modelling Language

- Statistical FSMs still done in assembly language
  - Build machines by manipulating arcs and states
  - For training,
    - get the weights by some exogenous procedure and patch them onto arcs
    - you may need extra training data for this
    - you may need to devise and implement a new variant of EM
- Would rather build models **declaratively**
  - $((a^{*.7}\ b)\ +_{.5}\ (ab^{*.6})) \circ \mathrm{repl}_{.9}((a{:}(b +_{.3} \varepsilon))^*,L,R)$

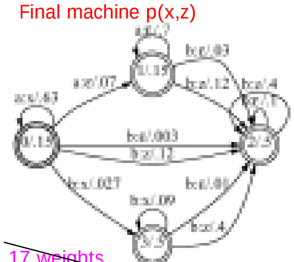
Outline

1. What can you build with parameterized FSMs?
2. How do you train them?

Hint: Make the finite-state machinery do the work.

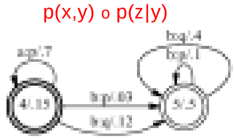
How Many Parameters?

Final machine p(x,z)

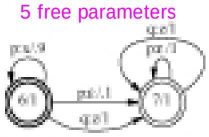


~~17 weights~~
~~– 4 sum-to-one constraints~~
~~= 13 apparently free parameters~~

But really I built it as
p(x,y) o p(z|y)



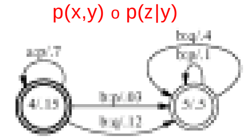
5 free parameters



1 free parameter

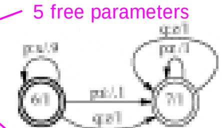
How Many Parameters?

But really I built it as
p(x,y) o p(z|y)



5 free parameters

Even these 6 numbers could be tied ... or derived by formula from a smaller parameter set.



1 free parameter

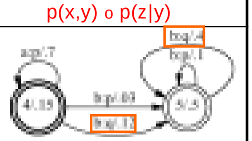
How Many Parameters?

But really I built it as
p(x,y) o p(z|y)

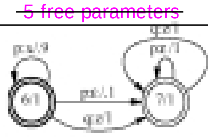
Really I built this as

$(a{:}p)^*_{.7}\ (b{:}\ (p +_{.2} q))^*_{.5}$

3 free parameters



~~5 free parameters~~



1 free parameter

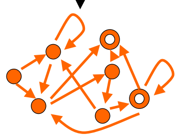
Training a Parameterized FST

Given: an expression (or code) to build the FST from a parameter vector θ

1. Pick an initial value of θ
2. Build the FST – implements fast prob. model
3. Run FST on some training examples to compute an objective function F(θ)
4. Collect E-counts or gradient ∇F(θ)
5. Update θ to increase F(θ)
6. Unless we converged, return to step 2

Training a Parameterized FST

… x_3 x_2 x_1

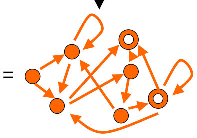
T =  (our current FST, reflecting our current guess of the parameter vector)

… y_3 y_2 y_1

At each training pair (x_i, y_i), collect E counts or gradients that indicate how to increase $p(x_i, y_i)$.

What are x_i and y_i?

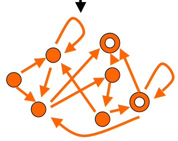
x_i

T =  (our current FST, reflecting our current guess of the parameter vector)

y_i

What are x_i and y_i?

x_i = banana

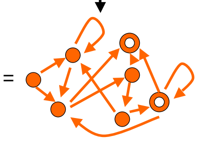
T =  (our current FST, reflecting our current guess of the parameter vector)

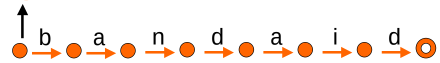
y_i = bandaid

What are x_i and y_i?

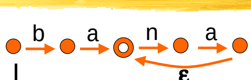
x_i = 

fully supervised

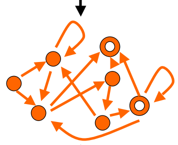
T =  (our current FST, reflecting our current guess of the parameter vector)

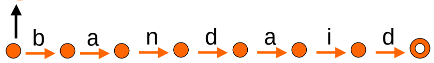
y_i = 

What are x_i and y_i?

x_i = 

loosely supervised

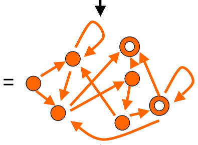
T =  (our current FST, reflecting our current guess of the parameter vector)

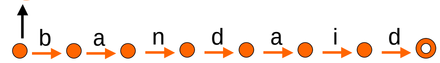
y_i = 

What are x_i and y_i?

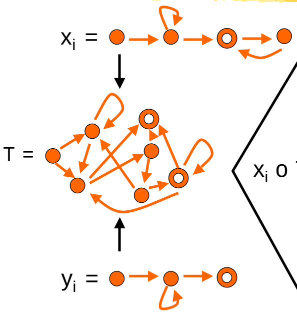
x_i = Σ^* = 

unsupervised, e.g., Baum-Welch.
Transition seq x_i is hidden
Emission seq y_i is observed

T =  (our current FST, reflecting our current guess of the parameter vector)

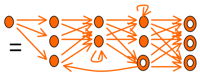
y_i = 

Building the Trellis



COMPOSE to get **trellis**:

x_i o T o y_i =

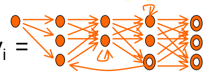


Extracts paths from T that are compatible with (x_i, y_i).

Tends to unroll loops of T, as in HMMs, but not always.

Summing the Trellis

x_i o T o y_i =



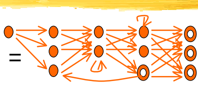
Extracts paths from T that are compatible with (x_i, y_i).
Tends to unroll loops of T, as in HMMs, but not always.

Let t_i = total probability of all paths in trellis = p(x_i, y_i) x_i, y_i are regexps (denoting strings or sets of strings) This is what we want to increase!

How to compute t_i?
If acyclic (exponentially many paths): dynamic programming.
If cyclic (infinitely many paths): solve sparse linear system.

Summing the Trellis

x_i o T o y_i =



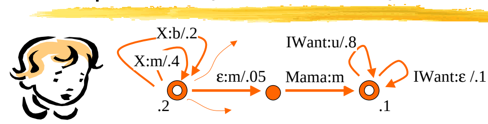
Let t_i = total probability of all paths in trellis = p(x_i, y_i). This is what we want to increase!

Remark: In principle, FSM minimization algorithm already knows how to compute t_i, although not the best method.

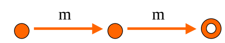
minimize (epsilonify (x_i o T o y_i)) = ◎ t_i

replace all arc labels with ε

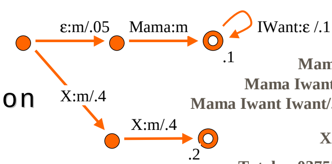
Example: Baby Think & Baby Talk



observe talk



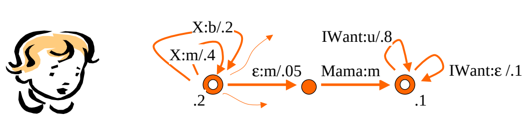
recover think, by composition



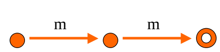
Mama/.005
Mama Iwant/.0005
Mama Iwant Iwant/.00005
XX/.032
Total = .0375555555

Joint Prob. by Double Composition

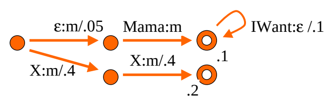
think



talk



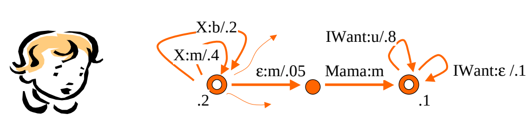
compose



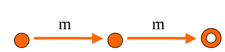
p(Σ* : mm) = .0375555 = sum of paths

Joint Prob. by Double Composition

think



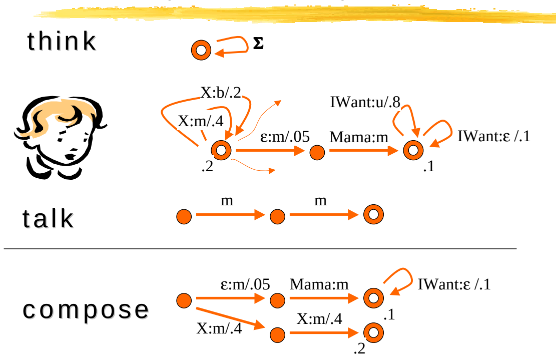
talk



compose

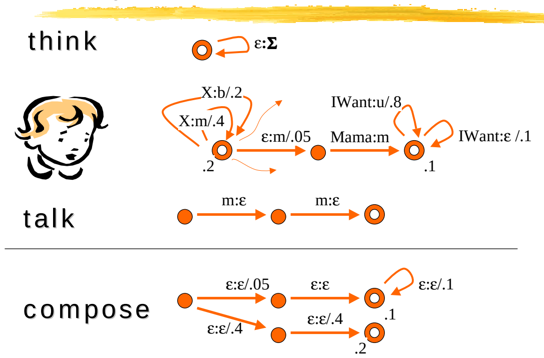
p(Σ* : mm) = .0005 = sum of paths

Joint Prob. by Double Composition



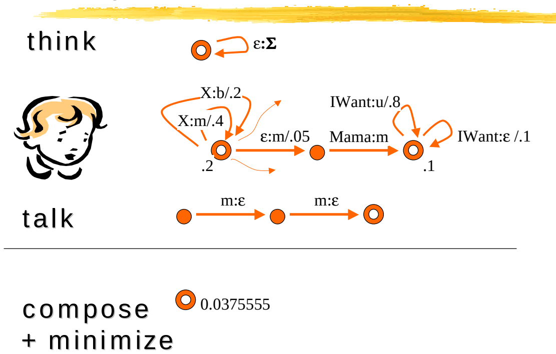
p(Σ* : mm) = .0375555 = sum of paths

Summing Over All Paths



p(Σ* : mm) = .0375555 = sum of paths

Summing Over All Paths



p(Σ* : mm) = .0375555 = sum of paths

Where We Are Now

"minimize (epsilonify (x_i o T o y_i))" = ●—ε/t_i→◎
obtains t_i = sum of trellis paths = $p(x_i, y_i)$.

Want to change parameters to make t_i increase.

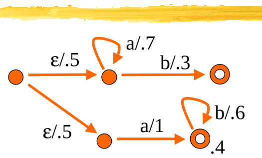
a vector

Solution: Annotate every probability with bookkeeping info.
So probabilities know how they depend on parameters.

Then the probability t_i will know, too!
It will emerge annotated with info about how to increase it.

The machine T is built with annotations from the ground up.

Probabilistic FSA

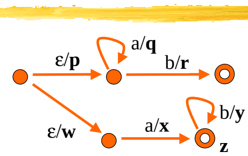


Example: ab is accepted along 2 paths
p(ab) = (.5 .7 .3) + (.5 .6 .4) = .225

Regexp: $(a^{*.7}\ b)\ +_{.5}\ (ab^{*.6})$

Theorem: Any probabilistic FSM has a regexp like this.

Weights Need Not be Reals

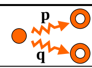
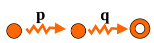
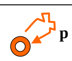
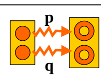


Example: ab is accepted along 2 paths
weight(ab) = (**p⊗q⊗r**) ⊕ (**w⊗x⊗y⊗z**)

If ⊗ ⊕ * satisfy "semiring" axioms,
the finite-state constructions
continue to work correctly.

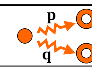
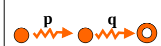
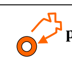
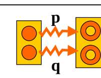
Semiring Definitions

Weight of a string is total weight of its accepting paths.

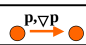
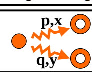
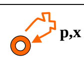
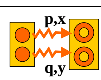
Union: ⊕		**p ⊕ q**
Concat: ⊗		**p ⊗ q**
Closure: *		**p***
Intersect, Compose: ⊗		**p ⊗ q**

The Probability Semiring

Weight of a string is total weight of its accepting paths.

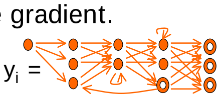
Union: ⊕		**p ⊕ q = p+q**
Concat: ⊗		**p ⊗ q = pq**
Closure: *		**$p^* = 1+p+p^2 + \ldots = (1-p)^{-1}$**
Intersect, Compose: ⊗		**p ⊗ q = pq**

The (Probability, Gradient) Semiring

Base case		**where ∇p is gradient**
Union: ⊕		**(p,x) ⊕ (q,y) = (p+q, x+y)**
Concat: ⊗		**(p,x) ⊗ (q,y) = (pq, py + qx)**
Closure: *		**$(p,x)^* = ((1-p)^{-1}, (1-p)^{-2}x)$**
Intersect, Compose: ⊗		**(p,x) ⊗ (q,y) = (pq, py + qx)**

We Did It!

- We now have a clean algorithm for computing the gradient.

x_i o T o y_i = 

Let t_i = total annotated probability of all paths in trellis
= ($p(x_i, y_i)$, $\nabla p(x_i, y_i)$). Aggregate over i (training examples).

How to compute t_i?
Just like before, when $t_i = p(x_i, y_i)$. But in new semiring.

If acyclic (exponentially many paths): dynamic programming.
If cyclic (infinitely many paths): solve sparse linear system.
Or can always just use minimize (epsilonify (x_i o T o y_i)).

An Alternative: EM

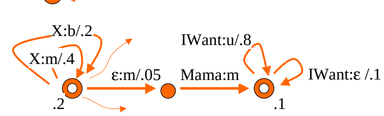
Would be easy to train probabilities
if we'd seen the paths the machine followed

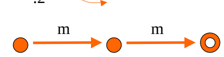
1. E-step: Which paths probably generated the observed data? (according to current probabilities)
2. M-step: Reestimate probabilities (or θ) as if those guesses were right
3. Repeat

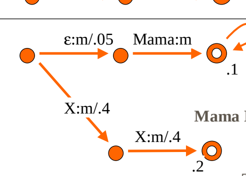
Guaranteed to converge to local optimum.

Baby Says mm

think x_i 

T 

talk y_i 

paths consistent with (x_i,y_i) 

Which Arcs Did We Follow?

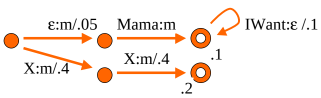
p(Mama : mm) = .005
p(Mama Iwant : mm) = .0005
p(Mama Iwant Iwant : mm) = .00005 etc.
p(XX : mm) = .032

p(mm) = p(Σ* : mm) = .0375555 = sum of *all* paths

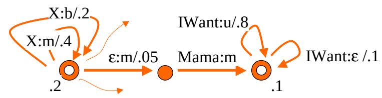
p(Mama | mm) = .005/.037555 = 0.13314
p(Mama Iwant | mm) = .0005/.037555 = 0.01331
p(Mama Iwant Iwant | mm) = .00005/.037555 = 0.00133
p(XX | mm) = .032/.037555 = 0.85207

} relative probs.

paths consistent with (Σ*, mm)



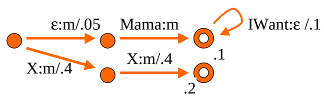
Count Uses of Original Arcs



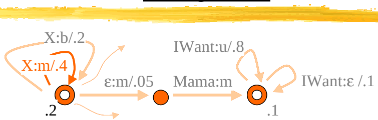
p(Mama | mm) = .005/.037555 = 0.13314
p(Mama Iwant | mm) = .0005/.037555 = 0.01331
p(Mama Iwant Iwant | mm) = .00005/.037555 = 0.00133
p(XX | mm) = .032/.037555 = 0.85207

} relative probs.

paths consistent with (Σ*, mm)



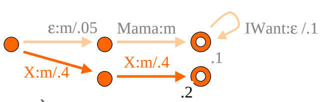
Count Uses of Original Arcs



p(Mama | mm) = .005/.037555 = 0.13314
p(Mama Iwant | mm) = .0005/.037555 = 0.01331
p(Mama Iwant Iwant | mm) = .00005/.037555 = 0.00133
p(XX | mm) = .032/.037555 = 0.85207

} relative probs.

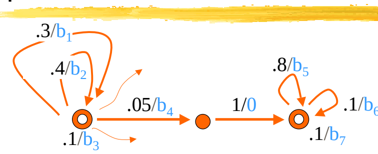
Expect 0.85207 × 2 traversals of original arc (on example Σ*, mm)



Associate a *value* with each arc we wish to track

Expected-Value Formulation

T =

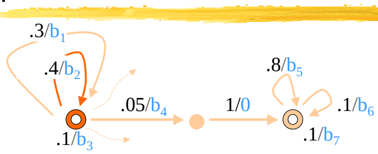


$b_1 = (1,0,0,0,0,0,0)$ $b_4 = (0,0,0,1,0,0,0)$ $b_7 = (0,0,0,0,0,0,1)$
$b_2 = (0,1,0,0,0,0,0)$ $b_5 = (0,0,0,0,1,0,0)$
$b_3 = (0,0,1,0,0,0,0)$ $b_6 = (0,0,0,0,0,1,0)$

Associate a *value* with each arc we wish to track

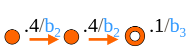
Expected-Value Formulation

T =



$b_1 = (1,0,0,0,0,0,0)$ $b_4 = (0,0,0,1,0,0,0)$ $b_7 = (0,0,0,0,0,0,1)$
$b_2 = (0,1,0,0,0,0,0)$ $b_5 = (0,0,0,0,1,0,0)$
$b_3 = (0,0,1,0,0,0,0)$ $b_6 = (0,0,0,0,0,1,0)$

$x_i \circ T \circ y_i$ =



has total value $b_2 + b_2 + b_3 = (0,2,1,0,0,0,0)$

Tells us the observed counts of arcs in T.

Associate a *value* with each arc we wish to track

Expected-Value Formulation

$x_i \circ T \circ y_i$ =

has total value $b_2 + b_2 + b_3 = (0,2,1,0,0,0,0)$

Tells us the observed counts of arcs in T.

But what if $x_i \circ T \circ y_i$ had multiple paths?
We want the **expected** path value for the E step of EM.
Some paths more likely than others.

expected value = Σ value(path) p(path | x_i, y_i)
= Σ value(path) p(path) / Σ p(path)

We'll arrange for t_i = (Σ p(path), Σ value(path) p(path))

$t_i = (\Sigma\ p(path), \Sigma\ value(path)\ p(path))$

The Expectation Semiring

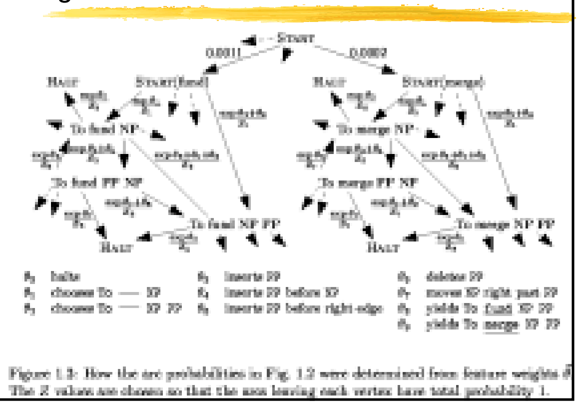
Base case	p, pv	where v is arc value
Union: ⊕	p,x / q,y	(p,x) ⊕ (q,y) = (p+q, x+y)
Concat: ⊗	p,x q,y	(p,x) ⊗ (q,y) = (pq, py + qx)
Closure: *	p,x	(p,x)* = $((1-p)^{-1}, (1-p)^{-2}x)$
Intersect, Compose: ⊗	p,x / q,y	(p,x) ⊗ (q,y) = (pq, py + qx)

same as before!

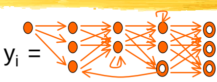
That's the algorithm!

- Existing mechanisms do all the work
- Keeps count of original arcs despite composition, loop unrolling, etc.
- Cyclic sums handled internally by the minimization step, which heavily uses semiring closure operation
- Flexible: can define arc values as we like
  - Example: Log-linear (maxent) parameterization
  - M-step: Must reestimate θ from feature counts (e.g., Iterative Scaling)
  - If arc's <u>weight</u> is $\exp(\theta_2+\theta_5)$, let its <u>value</u> be (0,1,0,0,1, ...)
  - Then total value of correct path for (x_i,y_i) – counts observed features
  - E-step: Needs to find expected value of path for (x_i,y_i)

Log-Linear Parameterization



Some Optimizations

x_i o T o y_i = 

Let t_i = total annotated probability of all paths in trellis
= $(p(x_i, y_i)$, bookkeeping information).

Exploit (partial) acyclicity

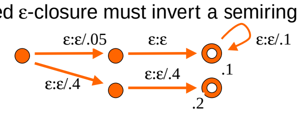
Avoid expensive vector operations

Exploit sparsity

Rebuild quickly after parameter update

Need Faster Minimization

- Hard step is the minimization:
  - Want total semiring weight of all paths
  - Weighted ε-closure must invert a semiring matrix



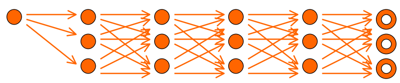
- Want to beat this! (takes $O(n^3)$ time)
- Optimizations exploit features of problem

All-Pairs vs. Single-Source

- For each q, r,
  ε-closure finds total weight of all q ⇝ r paths
- But we only need total weight of init ⇝ final paths
- Solve linear system instead of inverting matrix:
  - Let α(r) = total weight of init ⇝ r paths
  - $\alpha(r) = \Sigma_q\ \alpha(q) * weight(q \to r)$
  - $\alpha(init) = 1 + \Sigma_q\ \alpha(q) * weight(q \to init)$
- But still $O(n^3)$ in worst case

Cycles Are Usually Local

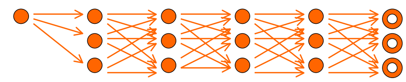
- In HMM case, Ti = $(\varepsilon \times x_i)$ o T o $(y_i \times \varepsilon)$ is an acyclic lattice:



- Acyclicity allows linear-time dynamic programming to find our sum over paths
- If not acyclic, first decompose into minimal cyclic components *(Tarjan 1972, 1981; Mohri 1998)*
  - Now full $O(n^3)$ algorithm must be run for several small n instead of one big n – and reassemble results
  - More powerful decompositions available *(Tarjan 1981); block-structured matrices*

Avoid Semiring Operations

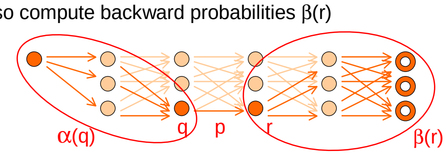
- Our semiring operations aren't O(1)
  - They manipulate vector values
- To see how this slows us down, consider HMMs:



- Our algorithm computes sum over paths in lattice.
  - If acyclic, requires a forward pass only.
- **Where's backward pass?**
- What we're pushing forward is (p,v)
  - Arcs v go forward to be downweighted by later probs, instead of probs going backward to downweight arcs.
  - The vector v rapidly loses sparsity, so this is slow!

Avoid Semiring Operations

- We're already computing forward probabilities α(q)
- Also compute backward probabilities β(r)



- Total probability of paths through this arc = α(q) * p * β(r)
- **E**[path value] = $\Sigma_{q,r}$ (α(q) * p(q → r) * β(r)) * value(q → r)
- Exploits structure of semiring
- Now α, β are probabilities, not vector values

Avoid Semiring Operations

- Now our linear systems are over the reals:
  - Let α(r) = total weight of init ⇝ r paths
  - α(r) = Σ_q α(q) * weight(q → r)
  - α(init) = 1 + Σ_q α(q) * weight(q → init)
- Well studied! Still $O(n^3)$ in worst case, but:
  - Proportionately faster for sparser graph
    - O(|states| |arcs|) by iterative methods like conj. gradient
    - Usually |arcs| << $|states|^2$
  - Approximate solutions possible
    - Relaxation *(Mohri 1998)* and back-relaxation *(Eisner 2001);* or stop iterative method earlier
  - Lower space requirement: O(|states|) vs. $O(|states|^2)$

Fast Updating

1. Pick an initial value of θ
2. Build the FST – implements fast prob. model

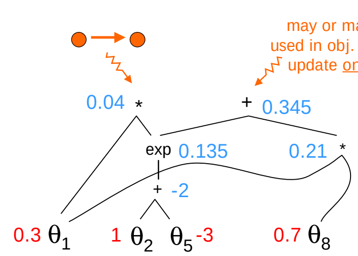
   …

6. Unless we converged, return to step 2

- But step 2 might be slow!
- Recompiles the FST from its parameterized regexp, using the new parameters θ.
- This involves a lot of structure-building, not just arithmetic
  - Matching arc labels in intersection and composition
  - Memory allocation/deallocation
  - Heuristic decisions about time-space tradeoffs

Fast Updating

- Solution: Weights remember underlying formulas
- A weight is a pointer into a formula DAG



Each node caches its current value

When (some) parameters are updated, invalidate (some) caches

Similar to a heap
Allows approximate updates

The Sunny Future

- Easy to experiment with interesting models.
- Change a model = edit declarative specification
- Combine models = give a simple regexp
- Train the model = push a button
- Share your model = upload to archive
- Speed up training = download latest version
  (conj gradient, pruning ...)
- Avoid local maxima = download latest version
  (deterministic annealing ...)
- p.s. Expectation semirings extend naturally to context-free case, e.g., Inside-Outside algorithm.

Marrying Two Finite-State Traditions

Classic stat models & variants ⇒ *simple FSMs*

HMMs, edit distance, sequence alignment, *n*-grams, segmentation

Trainable from data

Expert knowledge ⇒ *hand-crafted FSMs*

Extended regexps, phonology/morphology, info extraction, syntax ...

Tailored to task

Tailor model, then train end-to-end

Design complex finite-state model for task

Any extended regexp

Any machine topology; epsilon-cycles ok

Parameterize as desired to make it probabilistic

Combine models freely, tying parameters at will

Then find best param values from data (by EM or CG)

Ways to Improve Toolkit

- Experiment with other learning algs …
  - Conjugate gradient is a trivial variation; should be faster
  - Annealing etc. to avoid local optima
- Experiment with other objective functions …
  - Trivial to incorporate a Bayesian prior
  - Discriminative training: maximize p(y | x), not p(x,y)
- Experiment with other parameterizations …
  - Mixture models
  - Maximum entropy (log-linear):
    track expected feature counts, not arc counts
- Generalize more: Incorporate graphical modelling

Some Applications

- Prediction, classification, generation; more generally, “filling in of blanks”
  - Speech recognition
  - Machine translation, OCR, other noisy-channel models
  - Sequence alignment / Edit distance / Computational biology
  - Text normalization, segmentation, categorization
  - Information extraction
  - Stochastic phonology/morphology, including lexicon
  - Tagging, chunking, finite-state parsing
  - Syntactic transformations (smoothing PCFG rulesets)
- Quickly specify & combine models
- Tie parameters & train end-to-end
- Unsupervised, partly supervised, erroneously supervised

FIN

that’s all folks

(for now)

wish lists to **eisner@cs.jhu.edu**