Training Finite-State Transducer Cascades with Carmel

Kevin Knight, October 2009

Summary

This document describes how to train arbitrary cascades of finite-state machines on end-to-end data. The Carmel toolkit implements both EM (expectation-maximization) training and Bayesian Chinese Restaurant Process training.

It presumes familiarity with finite-state machines for natural language processing, and with basic Carmel functionality. If you need that background, see **A Primer on Finite-State Software for Natural Language Processing** (Knight & Al-Onaizan, 1999). It also presumes some familiarity with unsupervised learning.

The Carmel software was written by Jonathan Graehl, and it can be downloaded at http://www.isi.edu/licensed-sw/carmel/.

Contents

1. EM training of finite-state transducers (FSTs).
2. EM training of finite-state acceptors (FSAs).
3. EM training of cascades of finite-state machines on end-to-end data.
4. Sample applications.
5. Training FST + FST cascades.
6. Bayesian training of FST cascades on end-to-end data.

==========

1. EM training of finite-state transducers (FSTs).

Suppose we are modeling how English phoneme sequences (like "AY S K R IY M", for "ice cream") get borrowed into Japanese phoneme sequences (like "A I S U K U R I I M U"). We can develop a "generative story" like this:

Given: English phoneme sequence $e_1 \ldots e_n$
Produce: Japanese phoneme sequence

For each English phoneme token ei:
- Select a translation of e_i into j, according to $P(j \mid e_i)$,
  where j is some Japanese phoneme sequence of length 1 or 2.
- Output j

We then construct an FST that captures this story. In this case, the device takes an English phoneme (like "AY") and probabilistically maps it to every Japanese phoneme and to every sequence of two Japanese phonemes:

```
%%%%% Filename: epron-jpron.fst %%%%%
```

```
S
(S (S "AY" "A"))          # AY → A
(S (S "AY" "B"))          # AY → B
(S (S "AY" "D"))          # AY → D
(S (S "AY" "E"))          # AY → E
   …
(S (S1 "AY" "A"))         # AY → A A
(S1 (S *e* "A"))
(S (S2 "AY" "A"))         # AY → A B
(S2 (S *e* "B"))
   …
```

We add similar transitions for the other English phonemes. This FST embodies the $P(j \mid e_i)$ table -- every entry in that table has a corresponding transition in the FST. For efficiency, we can restrict the Japanese sequences to those actually observed in data (e.g., "Z" "B" will never be seen). The tutorial directory (sample-training/) has a toy version of this FST, with just a few phonemes included. Here is how to get statistics on the size of this FST:

```
% carmel -c epron-jpron.fst
Number of states in result: 57
Number of arcs in result: 154
```

Note that there are 154 transitions. The FST is unweighted and will map anything to anything with equal likelihood, so we need to train it on some data. We collect sample input/output pairs:

```
%%%%% Filename: epron-jpron.data %%%%%
"L" "IY" "N"        # lean
"R" "I" "N"
"R" "AE" "N"        # ran
"R" "A" "N"
"F" "AE" "N"        # fan
"H" "A" "N"
"L "AY" "N"         # line
"L" "A" "I" "N"
```

There are 4 string pairs here, with English input and Japanese output on alternating lines. Notice that the last input/output pair is tricky, because the Japanese is longer than the English. Therefore, the English "L" could map to Japanese "L", or it could map to "L" "A" -- in fact, there are three ways our FST can explain this string pair.

Internally, Carmel's EM training will build a derivation lattice for each string pair and run forward-backward training those lattices.

The Carmel command for EM training is "-t":

```
% carmel -t epron-jpron.data epron-jpron.fst
```

The output of "carmel -t" is a trained WFST. To make the output WFST look nice for this tutorial, we can do this:

```
% carmel -t -HJ epron-jpron.data epron-jpron.fst
```

The "-HJ" tells Carmel to format the WFST using one transition per line. The result is the trained machine:

```
S
(S (S "IY" "I" 1))
(S (S "AE" "A" 1))
(S (S "AY" "A" 5.5484658491927e-05))
(S (S "AY" "I" 2.77423292459635e-05))
(S (S22 "AY" "A" 0.999916773012262))
(S (S "L" "R" 0.999986128835377))
(S (S26 "L" "R" 1.38711646229817e-05))
(S (S "N" "N" 0.999986128835377))
(S (S33 "N" "I" 6.93558231149089e-06))
(S (S39 "N" "I" 6.93558231149089e-06))
(S (S "R" "R" 1))
(S (S "F" "H" 1))
(S22 (S *e* "I" 1))
(S26 (S *e* "A" 1))
(S33 (S *e* "N" 1))
(S39 (S *e* "N" 1))
```

EM has moved the probabilities of almost all of the 154 transitions to zero (or near zero) probability, leaving only a few solid transitions. EM training has therefore also decided what to with the ambiguous fourth training example. It believes: "L" → "L", "AY" → "A" "I", "N" → "N", and it guessed this by examining the other (unambiguous) training examples. If you had lots more data, and a full-blown phoneme-to-phoneme FST, you would learn a larger, probabilistic mapping table like the one shown in Figure 1 of [Knight & Graehl, 97].

If Carmel encounters an input/output pair that cannot be explained by your FST, then it will print a warning, and then *skip that example* for counting purposes. **These warnings are very useful for debugging your FST.**

"-t" will print a lot of information as it is training, before giving you the trained WFST:

```
i=1 (rate=1): probability=2^-43.6883 per-output-symbol-perplexity(N=13)=2^3.36064 per-
example-perplexity(N=4)=2^10.9221 (new best)
Initial best start point ppx=2^10.9221
Time for iteration: 0 sec

i=2 (rate=1): probability=2^-2.6413 per-output-symbol-perplexity(N=13)=2^0.203177 per-
example-perplexity(N=4)=2^0.660324 (new best) (relative-perplexity-
ratio=1.78182949764097e-07), max{d(weight)}=1
Time for iteration: 0.001 sec

i=3 (rate=1): probability=2^-0.926396 per-output-symbol-perplexity(N=13)=2^0.0712613 per-
example-perplexity(N=4)=2^0.231599 (new best) (relative-perplexity-ratio=0.15705600497582),
max{d(weight)}=0.45
Time for iteration: 0 sec

i=4 (rate=1): probability=2^-0.0639431 per-output-symbol-perplexity(N=13)=2^0.0049187 per-
example-perplexity(N=4)=2^0.0159858 (new best) (relative-perplexity-
ratio=1.38776570330882e-06), max{d(weight)}=0.277941176470588
Time for iteration: 0.001 sec

i=5 (rate=1): probability=2^-0.000240146 per-output-symbol-perplexity(N=13)=2^1.84728e-05
per-example-perplexity(N=4)=2^6.00366e-05 (new best) (relative-perplexity-ratio=e^
265.267182430138), max{d(weight)}=0.021975596541674
Converged - maximum weight change less than 0.0001 after 5 iterations.
Time for iteration: 0 sec
```

```
Setting weights to model with lowest per-example-perplexity ( =
prod[modelprob(example)]^(-1/num_examples) = 2^(-log_2(p_model(corpus))/N) =
2^6.00366e-05
```

When we use EM to train an FST, Carmel maximizes the conditional probability of the output strings given the input strings. In this case, that is P(Japanese | English). Above, you can see the data probability increase with each EM iteration. P(data) actually approaches 1.0 in this case -- in the final iteration, P(data) = 2^-0.000240146.

There are several switches for controlling EM training. Here are some:

-M <n> Carmel runs at most <n> EM iterations.

-X <n> Stops EM when the perplexity ratio from one iteration to the next is smaller than <n>. By default, this is 0.999, but it's wise to move this up to 0.99999 if you are doing serious training. If you use "-X 1.0", then EM will run for the full maximum number of iterations (-M).

-? Carmel remembers all the derivation lattice skeletons in memory, so subsequent iterations are fast. This can speed up training considerably. Without "-?", Carmel works with one training example at a time, rebuilding its derivation lattice at each iteration, which means it uses very little memory.

You can also communicate with Carmel EM through the FST you supply. If you put probabilities on your FST transitions, it tells EM to start with those weights as its initial point. If you put an exclamation point (!) after any probability in your FST, it tells EM that the transition's probability is locked, and EM won't change it. You can also tie transition probabilities together by putting them in the same normalization group n, by adding !<m> after their probabilities, where m>0 is some integer that represents that group.

If you don't want to start with uniform probabilities, and if you don't want to supply a start point yourself, then EM can choose a random start point for you:

-1 Carmel picks a random point to begin EM training, assigning a random probability to each transition.

This is useful for getting out of saddle-point situations, where EM can't decide which way to go, so it just stays still. Another useful switch is:

-! <n> Carmel executes <n> random restarts and output the transducer with the best trained P(data).

This saves you the work of managing restarts yourself. Because random restarts often improve performance, you can fire up "-! 50" after you're satisfied that things are working well. Note that this exclamation point has nothing to do with the (other) exclamation point that locks transition weights.

==========

2. EM training of finite-state acceptors (FSAs).

Carmel only knows how to train FSTs, so when we want to train an FSA on observed strings $s_1 \ldots s_n$, we just pretend like every output string s_i was produced by an empty input string *e*. So we write our training data as a sequence of alternating lines, with odd lines empty (*e*). For example:

```
%%%%% Filename: cluster.data %%%%%

"_" "C" "A" "R" "M" "E" "L" "_" "O" "N" "L" "Y" "_" "K" "N" "O" "W" "S" "_"

"_" "H" "O" "W" "_"

"_" "T" "O" "_" "T" "R" "A" "I" "N" "_"
      ...
```

Here's an interesting FSA that we can train on English data like this. Notice that every transition has *e* input, so that our FSA (really an FST) will be able to explain the examples in the cluster.data file.

```
%%%%% Filename: cluster.fsa %%%%%

0
(0 (0a *e* "_"))

(c1 (c1a *e* "A"))
(c1 (c1a *e* "B"))
(c1 (c1a *e* "C"))
       ...
(c1 (c1a *e* "Z"))

(c2 (c2a *e* "A"))
(c2 (c2a *e* "B"))
(c2 (c2a *e* "C"))
       ...
(c2 (c2a *e* "Z"))

(c3 (c3a *e* "A"))
(c3 (c3a *e* "B"))
(c3 (c3a *e* "C"))
       ...
(c3 (c3a *e* "Z"))

(0a (0 *e* *e*))
(0a (c1 *e* *e*))
(0a (c2 *e* *e*))
(0a (c3 *e* *e*))

(c1a (0 *e* *e*))
(c1a (c1 *e* *e*))
(c1a (c2 *e* *e*))
(c1a (c3 *e* *e*))

(c2a (0 *e* *e*))
(c2a (c1 *e* *e*))
(c2a (c2 *e* *e*))
(c2a (c3 *e* *e*))

(c3a (0 *e* *e*))
(c3a (c1 *e* *e*))
(c3a (c2 *e* *e*))
(c3a (c3 *e* *e*))
```

There are three important sets of probabilities: English letter probabilities on transitions coming out of the c1 state, out of the c2 state, and out of the c3 state. There are also transition probabilities between the states -- if we've just output a c1-type letter, should we stay in c1 or move to c2 or c3? To explain the English data, EM will manipulate all these probabilities.

```
% carmel -t -HJ -! 10 cluster.data cluster.fsa

0
(0 (0a *e* "_" 1))

(c1 (c1a *e* "D" 0.11302668104506))
(c1 (c1a *e* "E" 0.192866731218656))
(c1 (c1a *e* "L" 0.0509984219621482))
(c1 (c1a *e* "N" 0.13932430058078))
(c1 (c1a *e* "R" 0.0908026399351757))
(c1 (c1a *e* "S" 0.0924648094821811))
(c1 (c1a *e* "T" 0.0922497273441855))

(c2 (c2a *e* "B" 0.0739410458054848))
(c2 (c2a *e* "C" 0.0642975967819267))
(c2 (c2a *e* "M" 0.0831253579183909))
(c2 (c2a *e* "S" 0.0955582689506325))
(c2 (c2a *e* "T" 0.203897861423813))
(c2 (c2a *e* "W" 0.0872317935150973))

(c3 (c3a *e* "A" 0.212722696932441))
(c3 (c3a *e* "E" 0.146480864691274))
(c3 (c3a *e* "H" 0.118555785451024))
(c3 (c3a *e* "I" 0.191795687202824))
(c3 (c3a *e* "O" 0.201663674862103))
(c3 (c3a *e* "U" 0.0578441269070783))

(0a (0 *e* *e* 0.0896584819643285))
(0a (c2 *e* *e* 0.623511466798521))
(0a (c3 *e* *e* 0.286783668236858))
(c1a (0 *e* *e* 0.529971677705547))
(c1a (c1 *e* *e* 0.386887330724283))
(c2a (c2 *e* *e* 0.0646454803752674))
(c2a (c3 *e* *e* 0.889354631315224))
(c3a (0 *e* *e* 0.124860108792928))
(c3a (c1 *e* *e* 0.597520721801701))
(c3a (c2 *e* *e* 0.107944386642835))
(c3a (c3 *e* *e* 0.169674782762536))
```

I have only shown the highly weighted transitions here. You can see how EM divided the letters into groups. The "-!" random restart is necessary here, because this is a saddle-point otherwise. If you run this command, you will get different clusters, because of randomness.

==========

3. EM training of cascades of finite-state machines on end-to-end data.

Sometimes it is easier to define a sequence (or cascade) of machines for your problem, instead of one big spaghetti machine.

For example, the FSA above is a little messy. A simpler way to define the same thing is to make an FSA that generates sequences of symbols "c1" and "c2" and "c3", and then make a separate FST that turns those sequences into English.

Our new FSA is a fully-connected bigram model:

```
0
(0 (0 *e* "space"))
(0 (c1 *e* "c1"))
(0 (c2 *e* "c2"))
(0 (c3 *e* "c3"))
(c1 (0 *e* "space"))
(c1 (c1 *e* "c1"))
(c1 (c2 *e* "c2"))
(c1 (c3 *e* "c3"))
(c2 (0 *e* "space"))
(c2 (c1 *e* "c1"))
(c2 (c2 *e* "c2"))
(c2 (c3 *e* "c3"))
(c3 (0 *e* "space"))
(c3 (c1 *e* "c1"))
(c3 (c2 *e* "c2"))
(c3 (c3 *e* "c3"))
```

Our new "substitute" FST has only one state:

```
0
(0 (0 "space" "_"))
(0 (0 "c1" "A"))
(0 (0 "c1" "B"))
        ...
(0 (0 "c1" "Z"))
(0 (0 "c2" "A"))
(0 (0 "c2" "B"))
        ...
(0 (0 "c2" "Z"))
(0 (0 "c3" "A"))
(0 (0 "c3" "B"))
        ...
(0 (0 "c3" "Z"))
```

Both of these models are simple and easy to understand. When cascaded (FSA + FST), they produce English according to the probabilities in the machines. We'd like to both machines' probabilities on observed English. Here is the command:

```
% carmel --train-cascade -HJ cluster.data cat.fsa spellout.fst
```

Notice the "--train-cascade" command to run EM. What will Carmel output? Unlike before, we now want **two** trained machines out -- the trained version of cat.fsa and the trained version of spellout.fst. Carmel therefore creates ".trained" versions of those machine files. If you look in your directory after running this command, you will see two new files:

```
%%%%% Filename: cat.fsa.trained %%%%%

%%%%% Filename: spellout.fst.trained %%%%%
```

How does Carmel work internally? It composes the machines into a bigger machine X, but keeps the transitions of the two machines separate from one another. It also ties transitions in X that need to be tied. After running EM on this machine, it de-composes the X machine back into the original machines, but with the learned weights -- these are the ".trained" machines that are output back to you.

We can now use the trained cascade to get a Viterbi tagging for any English text -- every letter will be assigned either cat1 or cat2. To do this, we need to build a version of cat.fsa.trained that removes the *e* from the transition inputs -- let's call it cat.fsa.trained.noe. For batch Viterbi processing, our data should also appear without blank lines (cluster.data.noe). Carmel can help us create that stuff:

```
% carmel --project-right --project-identity-fsa -HJ cat.fsa.trained > cat.fsa.trained.noe
% awk 'NF>0' cluster.data > cluster.data.noe
```

To Viterbi-decode with our trained cascade, we type:

```
% cat cluster.data.noe | carmel -qbsriWIEk 1 cat.fsa.trained.noe spellout.fst.trained
```

The "-b" switch tells Carmel to process the lines in cluster.data.noe one at a time, and the "-r" switch tells Carmel that those lines should be inserted on the right side of the cascade, not the left.

The first three lines of cluster.data.noe are:

- `"_" "C" "A" "R" "M" "E" "L" "_" "O" "N" "L" "Y" "_" "K" "N" "O" "W" "S" "_"`
- `"_" "H" "O" "W" "_"`
- `"_" "T" "O" "_" "T" "R" "A" "I" "N" "_"`

The first three lines of our Viterbi decoding call are:

- `"space" "c3" "c1" "c2" "c3" "c1" "c2" "space" "c1" "c2" "c2" "c2" "space" "c3" "c3" "c1" "c2" "c2" "space"`
- `"space" "c3" "c1" "c2" "space"`
- `"space" "c3" "c1" "space" "c3" "c3" "c1" "c1" "c2" "space"`

==========

4. Sample applications.

Many interesting problems can be attacked with finite-state cascades. Here are two:

Unsupervised part-of-speech tagging. Our observed data is raw text. The FSA is a tag n-gram model, and the FST is a single-state substituter of words for tags. The FSA is fully connected, and FST is constrained by a provided dictionary. We have to learn both machines' parameters in an unsupervised way. Relevant files and commands are:

```
%%%%% Filename: tagging.data %%%%%
%%%%% Filename: tagging.key %%%%%
%%%%% Filename: tagging.fsa %%%%%
%%%%% Filename: tagging.fst %%%%%

% carmel --train-cascade -HJ tagging.data tagging.fsa tagging.fst
% carmel --project-right --project-identity-fsa -HJ tagging.fsa.trained > tagging.fsa.trained.noe
% awk 'NF>0' tagging.data > tagging.data.noe
```

```
% cat tagging.data.noe | carmel -qbsriWIEk 1 tagging.fsa.trained.noe tagging.fst.trained
```

Letter substitution decipherment. Our observed data is a coded sequence that has been produced like this -- someone took an English message and replaced every letter according to a substitution table. The FSA is an English letter n-gram model, and the FST is a single-state substituter. The FST is fully-connected, with 26 x 26 transitions, initially all of equal weight. In this case the FSA is trained on English and the probabilities are locked before we run "--train-cascade". Relevant files and commands are:

```
%%%%% Filename: cipher.data %%%%%
%%%%% Filename: cipher.key %%%%%
%%%%% Filename: cipher.fsa %%%%%
%%%%% Filename: cipher.fst %%%%%

% carmel --train-cascade -HJ cipher.data cipher.wfsa cipher.fst
% carmel --project-right --project-identity-fsa -HJ cipher.wfsa > cipher.wfsa.noe
% awk 'NF>0' cipher.data > cipher.data.noe
% cat cipher.data.noe | carmel -qbsriWIEk 1 cipher.wfsa.noe cipher.fst.trained
```

You can try these out -- the files are in the tutorial directory. The correct answers can be found in the files tagging.key and cipher.key. For both problems, results will be improved by random restarts and more iterations.

==========

5. Training FST + FST cascades.

So far, we have talked about training FSA + FST cascades, but we can also train FST + FST cascades. Suppose we have a first FST that probabilistically copies or deletes letters:

```
%%%%% Filename: delete.fst %%%%%

0
(0 (0 "a" "a"))
(0 (0 "a" *e*))
(0 (0 "b" "b"))
(0 (0 "b" *e*))
(0 (0 "c" "c"))
(0 (0 "c" *e*))
```

Suppose we have a second FST probabilistically transforms letters, independent of context:

```
%%%%% Filename: transform.fst %%%%%

0
(0 (0 "a" "a"))
(0 (0 "a" "b"))
(0 (0 "a" "c"))
(0 (0 "b" "a"))
(0 (0 "b" "b"))
(0 (0 "b" "c"))
(0 (0 "c" "a"))
(0 (0 "c" "b"))
(0 (0 "c" "c"))
```

Finally, suppose we observe the following data:

```
%%%%% Filename: deltrans.data %%%%%

"a" "b" "c"      # "a" "b" "c" → "b" "a"
"b" "a"
"a" "b" "c"      # "a" "b" "c" → "a" "c"
"a" "c"
"a" "c"          # "a" "c"     → "b" "c"
"b" "c"
"a" "c"          # "a" "c"     → "b" "c"
"b" "c"
```

We now run EM training:

```
% carmel --train-cascade -X 0.99999 -! 10 deltrans.data delete.fst transform.fst
```

Notice the 10 random restarts ("-! 10") and the stringent convergence request ("-X 0.99999" instead of the default 0.999). This command generates the following ".trained" versions of our machines:

```
%%%%% Filename: delete.fst.trained %%%%%

0
(0 (0 "a" "a" 0.75))
(0 (0 "a" *e* 0.25))
(0 (0 "b" "b" 1))
(0 (0 "b" *e* 6.8615573360298e-18))
(0 (0 "c" "c" 0.75))
(0 (0 "c" *e* 0.25))

%%%%% Filename: transform.fst.trained %%%%%

0
(0 (0 "a" "a" 1.97761685976632e-23))
(0 (0 "a" "b" 1))
(0 (0 "b" "a" 1))
(0 (0 "b" "b" 2.39559820862215e-21))
(0 (0 "b" "c" e-187.527164826805))
(0 (0 "c" "a" 4.57594884665706e-18))
(0 (0 "c" "c" 1))
```

For the first FST, EM learned that input symbols "a" and "c" can sometimes be deleted (0.25 chance), while "b" is never deleted. EM also learned the following transformations for the second FST: "a" goes to "b", "b" goes to "a", and "c" goes to "c".

Because of the random restarts, you may get a different answer when you run it. For example, the following solution always deletes input "b" tokens, leaving the remaining "a" and "c" tokens to transform themselves into the observed output.

```
%%%%% Filename: delete.fst.trained %%%%%

0
(0 (0 "a" "a" 1))
(0 (0 "a" *e* 1.36958138882441e-18))
(0 (0 "b" "b" 2.88391710752266e-18))
(0 (0 "b" *e* 1))
(0 (0 "c" "c" 1))
```

```
(0 (0 "c" *e* 7.23771649369207e-20))

%%%%% Filename: transform.fst.trained %%%%%

0
(0 (0 "a" "a" 0.25))
(0 (0 "a" "b" 0.75))
(0 (0 "b" "a" 0.999726408340245))
(0 (0 "b" "b" 0.000273591658524948))
(0 (0 "b" "c" 1.22743061972586e-12))
(0 (0 "c" "a" 0.25))
(0 (0 "c" "c" 0.75))
```

EM almost always has local minima and multiple solutions. You can look at EM's trace to check what P(data) numbers you are getting.

Note that one string in deltrans.data was observed multiply (namely, "a" "c" goes to "b" "c", which was seen twice). If a string is observed thousands of times, it is inefficient for Carmel EM to process it thousands of times, building the same derivation lattice and collecting the same counts. Carmel therefore accepts a data format that is more compact when observations are repeated. This format uses triples of lines (frequency/input/output) instead of pairs of lines (input/output):

```
%%%%% Filename: deltrans.data.compact %%%%%

1                  # "a" "b" "c" → "b" "a" was observed 1 time
"a" "b" "c"
"b" "a"
1                  # "a" "b" "c" → "a" "c" was observed 1 time
"a" "b" "c"
"a" "c"
2                  # "a" "c" → "b" "c"     was observed 2 times
"a" "c"
"b" "c"
```

The following two commands will yield exactly the same output, but the second command will require fewer computations to get there:

```
% carmel --train-cascade deltrans.data         delete.fst transform.fst
% carmel --train-cascade deltrans.data.compact delete.fst transform.fst
```

==========

6. Bayesian training of FST cascades on end-to-end data.

Recently, Bayesian methods have become popular in natural language processing. Here, I don't mean Bayesian as in "Noisy channel FSA+FST applications that use Bayes Rule" (though those are cool!), but rather as in "Methods of inference that integrate over uncertainty about the values that parameters take on".

Please see **Bayesian Inference with Tears** [Knight, 2009] for background.

Carmel packages up what is called "cache model training" in **Bayesian Inference with Tears**. This packaging lets you type "--crp" (Chinese Restaurant Process) instead of "--train-cascade", and you get

out ".trained" transducers, as before. If you do "--crp" instead of "--train-cascade", you may experience better accuracy results on your task. So, in current Carmel, Bayesian inference is a drop-in replacement for EM. You supply the transducer cascade (as before) and the training data (as before), and you get back ".trained" transducers (as before).

Recall our data and machine files for the unsupervised part-of-speech tagging problem:

```
%%%%% Filename: tagging.data %%%%%
%%%%% Filename: tagging.fsa %%%%%
%%%%% Filename: tagging.fst %%%%%
```

Here is how we do Bayesian training:

```
% carmel --crp -M 6000 tagging.data tagging.fsa tagging.fst
```

This command generates tagging.fsa.trained and tagging.fst.trained, and you can use them to get Viterbi decodings of the data.

What is going on internally? Carmel is running many iterations of blocked Gibbs sampling, as described in "Bayesian Inference with Tears". It kicks things off with a randomly chosen sample through the derivation lattices.

For the end user, the blocked part may be important. It means that you cannot train "--crp" on a single huge sequence. Your data needs to naturally break into separate training lines (blocks). Natural breaks may be sentences, or words, or arbitrary sequences. EM, of course, can train on one big sequence.

A word of warning: because of randomness in the initialization and re-sampling, you will get different results every time you run. You may find yourself excited about runs that yield high task accuracy and depressed about runs that yield low task accuracy.

There are many knobs to twist in Bayesian inference. Here are some of them:

user-supplied probabilities Probabilities that you supply in your pre-trained devices are interpreted differently by "--crp". They do not define a start point for the sampler, but rather they represent a user-supplied prior base distribution. Lacking data, the training will go with these base distribution values. As data increases, training will move carefully away from them. When there are vast amounts of data, training may forget the base distribution values entirely.

--priors=0.01,0.001 Accepts an alpha concentration parameter for each transducer in the cascade. (The alphas are separated by commas). A high alpha means: always trust the base distribution. A low alpha means: dis-trust the base distribution. If you put alpha at 10,000, then your ".trained" machine will come out just like the original. The default alpha is 0.01.

-M Number of Gibbs sampling iterations. This is typically much larger than -M for --train-cascade.

--high-temp=<n>, --low-temp=<n> Controls annealing in the Gibbs sampler. When n=1, a new sample is chosen with probability proportional to the new corpus-wide derivation it induces. When n>1, the coin-flip for the new sample is made more random (i.e., we can easily flip to a weird new sample), and when n<1, the coin-flip is less random (i.e., we tend to stick with the highest-scoring new sample).

The sampler starts with the "--high-temp" value on the first Gibbs iteration, and it gradually reduces the randomness until it is using "--low-temp" on the final Gibbs iteration.

--burnin=<n> The Gibbs sampler collects whole counts off derivations that it samples. This command instructs Carmel not to collect counts off the first <n> samples.

--final-counts This command tells Carmel to collect counts only off the final derivation.

What happens internally when Carmel creates the ".trained" machines? It computes parameter values by taking into account both (1) what is observed in the generated samples, and (2) the base distribution and concentration parameters. That is why "--priors=10000" will cause the ".trained" machine to look just like the original machine, with its originally supplied base distribution.

==========

Questions? Please send questions to Jonathan Graehl (graehl@isi.edu) or Kevin Knight (knight@isi.edu).

==========

Commands found in "Training Finite-State Transducer Cascades with Carmel" (this document):

```
carmel -c epron-jpron.fst
carmel -t epron-jpron.data epron-jpron.fst
carmel -t -HJ epron-jpron.data epron-jpron.fst

carmel -t -HJ -! 10 cluster.data cluster.fsa
carmel --train-cascade -HJ cluster.data cat.fsa spellout.fst
carmel --project-right --project-identity-fsa -HJ cat.fsa.trained > cat.fsa.trained.noe
awk 'NF>0' cluster.data > cluster.data.noe
cat cluster.data.noe | carmel -qbsriWIEk 1 cat.fsa.trained.noe spellout.fst.trained

carmel --train-cascade -HJ tagging.data tagging.fsa tagging.fst
carmel --project-right --project-identity-fsa -HJ tagging.fsa.trained > tagging.fsa.trained.noe
awk 'NF>0' tagging.data > tagging.data.noe
cat tagging.data.noe | carmel -qbsriWIEk 1 tagging.fsa.trained.noe tagging.fst.trained

carmel --train-cascade -HJ cipher.data cipher.wfsa cipher.fst
carmel --project-right --project-identity-fsa -HJ cipher.wfsa > cipher.wfsa.noe
awk 'NF>0' cipher.data > cipher.data.noe
cat cipher.data.noe | carmel -qbsriWIEk 1 cipher.wfsa.noe cipher.fst.trained

carmel --train-cascade -X 0.99999 -! 10 deltrans.data delete.fst transform.fst
carmel --train-cascade deltrans.data delete.fst transform.fst
carmel --train-cascade deltrans.data.compact delete.fst transform.fst

carmel --crp -M 6000 tagging.data tagging.fsa tagging.fst
```