Chapter 6

Text Input

6.1 Problem

In the last two chapters we looked at language models, and in your first homework you are building language models to enable the computer to guess the next work that the user will type. Now, what if we want the computer to actually correct typos? We can use a *hidden Markov model* (HMM) do to this. HMMs are useful for a lot of other things as well. Later in this unit, we'll look at speech recognition and optical character recognition. For your homework, you'll try converting Shakespearean English into modern English. But here, we'll focus on typo correction.

6.2 Hidden Markov Models

6.2.1 Definition

An HMM is an example of a generative model, which tries to model the probability not just of outputs given inputs, but the probability of both inputs and outputs. They are sort of like investigators at a crime scene: instead of trying to reason directly from evidence (input) to a culprit (output), it is natural to imagine, for various possible culprits, how the crime may have played out to yield the observed evidence, and then decide which of those various scenarios is the most plausible.

Similarly, instead of thinking directly about finding the most probable sequence of true characters, we think about how the observed sequence might have come to be. Let $\mathbf{w} = w_1 \cdots w_n$ be a sequence of observed characters and let $\mathbf{t} = t_1 \cdots t_n$ be a sequence of true characters that the user wanted to type. To distinguish between the two, we'll write observed characters in `typewriter font` and intended characters in sans-serif font.

We want to find the most probable $\mathbf{t}$ given $\mathbf{w}$, which is the same as the $\mathbf{t}$ that maximizes the joint probability:

$$\arg\max_{\mathbf{t}} P(\mathbf{t} \mid \mathbf{w}) = \arg\max_{t} P(\mathbf{t}, \mathbf{w}) \tag{6.1}$$

which we can rewrite as

$$P(\mathbf{t}, \mathbf{w}) = P(\mathbf{t})P(\mathbf{w} \mid \mathbf{t}). \tag{6.2}$$

The first term, $P(\mathbf{t})$, models things that a user might plausibly want to type, and can be described using a language model, for example, a bigram model:

$$P(\mathbf{t}) = p(t_1 \mid \texttt{<s>}) \times \left(\prod_{i=2}^{n} p(t_i \mid t_{i-1}) \right) \times p(\texttt{</s>} \mid t_n). \tag{6.3}$$

The second term, $P(\mathbf{w} \mid \mathbf{t})$, models how what the user wanted to type (**t**) becomes what they actually type (**w**). Let's call this the *typo model*. We choose a very simple definition for it:

$$P(\mathbf{w} \mid \mathbf{t}) = \prod_{i=1}^{n} p(w_i \mid t_i). \tag{6.4}$$

This is a *hidden Markov model* (HMM).

Suppose that we are given labeled data, like:

t	t	y	p	e
w	t	h	p	e

In this case, the HMM is easy to train: just count and divide for both the language model and the typo model.

But what we don't know how to do is *decode*: given **w**, what's the most probable **t**? It is *not* good enough to say that our best guess for t_i is

$$\hat{t}_i = \arg\max_t p(t \mid t_{i-1}) p(w_i \mid t) \qquad \text{(wrong)}.$$

Why? Imagine that the user types `thpe`. At the time he/she presses `h`, it seems like a very plausible letter to follow `t`. But when he/she types `pe`, it becomes clear that there was a typo. We now go *back* and realize that `h` was meant to be `y`. So what we really want is

$$\hat{\mathbf{t}} = \arg\max_{\mathbf{t}} P(\mathbf{t} \mid \mathbf{w}) = \arg\max_{\mathbf{t}} P(\mathbf{t}) P(\mathbf{w} \mid \mathbf{t}). \tag{6.5}$$

Naively, this would seem to require searching over all possible **t**, of which there are exponentially many. But below, we'll see how to do this efficiently, not just for HMMs but for a much broader class of models.

6.2.2 Decoding

Suppose that our HMM has the parameters shown in Table 6.1. (In reality, of course, there would be many more.)

Now, given the observed characters `thpe`, we can construct a FSA that generates all possible sequences of true characters for this sentence. See Figure 6.1. Here's how to construct this FSA (called a *lattice* or *trellis*) for an HMM. If the observed sequence is $\mathbf{w} = w_1 \cdots w_n$, and the alphabet size is $|\Sigma|$, then the states form a grid with $|\Sigma|$ rows and $n + 2$ columns. Call each state $q_{i,t}$ where $0 \leq i \leq n$ and $t \in \Sigma$. The meaning of each such state is that we are at time step i and the previous true character is t. The start state is $q_{0,\texttt{<s>}}$ and the accept state is $q_{n+1,\texttt{</s>}}$. There's a transition from every state $q_{i-1,t}$ to every state $q_{i,t'}$ with weight $p(t' \mid t) p(w_i \mid t')$.

$p(t' \mid t)$

	t					
t'	<s>	e	h	p	t	y
e	0.2	0.2	0.5	0.2	0.1	0.4
h	0.2	0.1	0	0.1	0.6	0
p	0.1	0.1	0	0.2	0	0.1
t	0.4	0.2	0.1	0.1	0.1	0.1
y	0.1	0.1	0	0.2	0.1	0
</s>	0	0.3	0.4	0.2	0.1	0.4

$p(w \mid t)$

	t				
w	e	h	p	t	y
e	1	0	0	0.5	0
h	0	0.5	0	0	0.25
p	0	0	1	0	0
t	0	0	0	0.5	0.25
y	0	0.5	0	0	0.5

Table 6.1: Example parameters of an HMM.

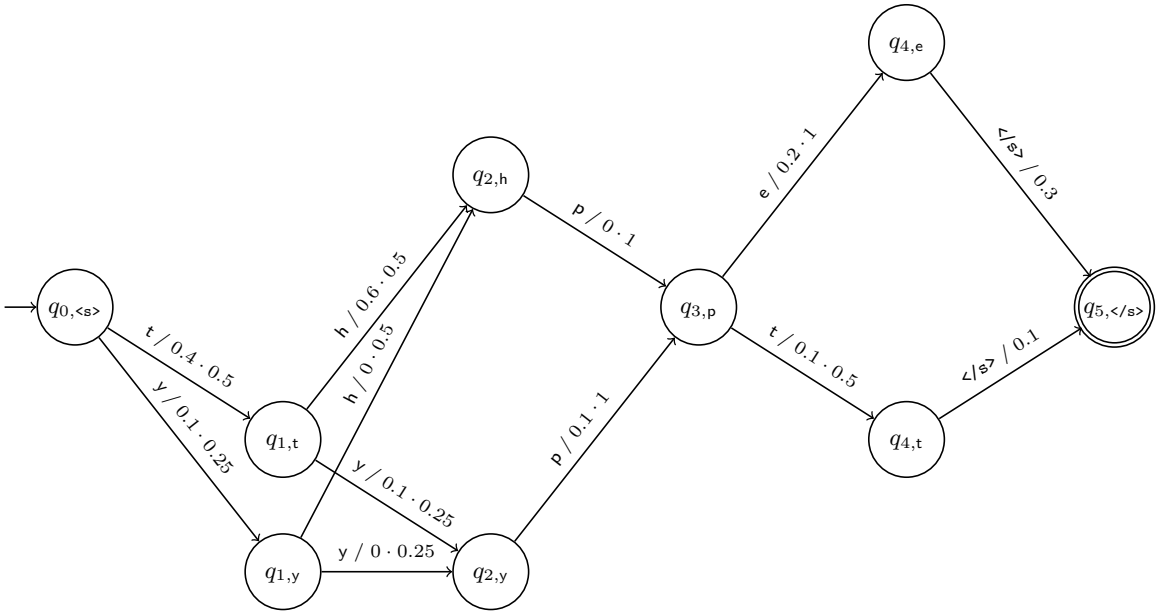


Figure 6.1: Weighted finite automaton for all possible true characters corresponding to the observed characters `thpe`.

Each path through this FSA corresponds to a possible true string **t** that the user may have meant to type, and the weight of each path is $P(\mathbf{t}, \texttt{thpe})$. Our goal is to find the highest-weight path through M. We can do this efficiently using the *Viterbi algorithm*, originally invented for decoding error-correcting codes. It works not only on FSAs of the form described above, but any acyclic FSA.

The Viterbi algorithm is a classic example of dynamic programming. We need to visit the states of M in *topological order*, that is, so that all the transitions go from earlier states to later states in the ordering. For the example above, the states are all named $q_{i,\sigma}$; we visit them in order of increasing i. For each state q, we want to compute the weight of the best path from the start state to q. This is easy, because we've already computed the best path to all of the states that come before q. We also want to record which incoming transition is on that path. The algorithm goes like this:

viterbi$[q_0] \leftarrow 1$
viterbi$[q] \leftarrow 0$ **for** $q \neq q_0$
for each state q' in topological order **do**
  for each incoming transition $q \rightarrow q'$ with weight p **do**
    if viterbi$[q] \times p >$ viterbi$[q']$ **then**
      viterbi$[q'] \leftarrow$ viterbi$[q] \times p$
      pointer$[q'] \leftarrow q$
    end if
  end for
end for

Then the maximum weight is viterbi$[q_f]$, where q_f is the final state.

Question 2. How do you use the pointers to reconstruct the best path?

6.3 Hidden Markov Models (Unsupervised)

Above, we assumed that the model was given to us. If we had *parallel data*, that is, data consisting of pairs of strings where one string has typos and the other string is the corrected version, then it's easy to train the model (just count and divide). This kind of data is called *complete* because it consists of examples of exactly the same kind of object that our model is defined on.

What if we have *incomplete* data? For example, what if we only have a collection of strings with typos and a collection of strings without typos, but the two collections are not parallel? We can still train the language model as before, but it's no longer clear how to train the typo model. To do this, we need a fancier (and slower) training method. We'll first look at a more intuitive version, and then a better version that comes with a mathematical guarantee.

The basic idea behind both methods is a kind of bootstrapping: start with some model, not necessarily a great one. For example,

$$p(w \mid t) = \begin{cases} \frac{2}{|\Sigma_t|+1} & \text{if } t = w \\ \frac{1}{|\Sigma_t|+1} & \text{if } t \neq w \end{cases}$$

where Σ_t is the alphabet of true characters.

Now that we have a model, we can correct the typos in the typo part of the training data. This gives us a collection of parallel strings, that is, pairs of strings where one string has typos and the

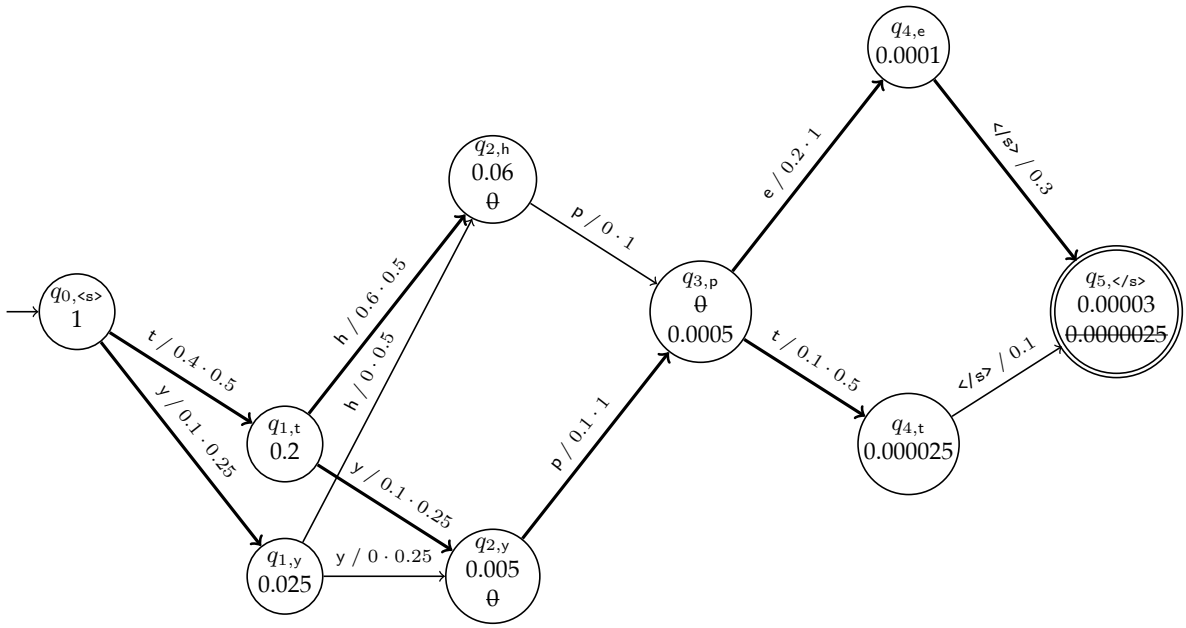


Figure 6.2: Example run of the Viterbi algorithm on the FSA of Figure 6.1. Probabilities are written inside states; probabilities that are not maximum are ~~struck out~~. The incoming edge that gives the maximum probability is drawn with a thick rule.

other string is its corrected version. The corrections might not be great, but at least we have parallel data.

Then, use the parallel data to re-train the model. This is easy (just count and divide). The question is, will this new model be better than the old one?

6.3.1 Hard Expectation-Maximization

"Hard" Expectation-Maximization is the "hard" version of the method we'll see in the next section. It's "hard" not because it's hard to implement but because when it corrects the typo part of the training data, it makes a hard decision; that is, it commits to one and only one correction for each string. We've seen already that we can do this efficiently using the Viterbi algorithm.

In pseudocode, hard EM works like this. Assume that we have a collection of N observed strings, $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots \mathbf{w}^{(N)}$.

initialize the model
repeat
  for each observed string: $\mathbf{w}^{(i)}$ **do** ▷ Viterbi E step
    compute the best correction, $\hat{\mathbf{t}} = \arg\max_{\mathbf{t}} P(\mathbf{t} \mid \mathbf{w}^{(i)})$
    for each position j **do**
      $c(\hat{\mathbf{t}}, \mathbf{w}_j^{(i)}) \mathrel{+}= 1$
    end for
  end for
  for all w, t **do** ▷ M step
    $p(w \mid t) = \frac{c(t,w)}{\sum_{w'} c(t,w')}$
  end for
until converged

In practice, this method is easy to implement, and can work very well. Let's see what happens when we try this on some real data. The corrected part is 50,000 lines from an IRC channel for Ubuntu technical support. (Actually this data has some typos, but we pretend it doesn't.) The typo part is another 50,000 lines with typos artificially introduced.

Using the initial model, we get corrections like:

```
observed: bootcharr? i recall is habdy.
guess:    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

But some are slightly less bad, like:

```
observed: Hello :)
guess:    I the t?
```

After a few iterations, these two lines have improved to:

```
observed: bootcharr? i recall is habdy.
guess:    tootcharr? i recath is hathe.
```

and

```
observed: Hello :)
guess:    I tho e)
```

So it's learned to recover the fact that letters usually stay the same, but it still introduces more errors than it corrects.

6.3.2 Expectation-Maximization

Above, we saw that the initial model sometimes corrects strings to all carets (^) and sometimes to strings with lots of variations on the word `the`. What if all the strings were of the former type? The model would forget anything about any true character other than ^, and it would get stuck in this condition forever.

More generally, when the model looks at a typo string, it defines a whole distribution over possible corrections of the string. For example, the initial model likes ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ best, but it also knows that just leaving everything alone is better than random characters. When we take only the model's best guess, we throw away potentially useful information.

More formally, if we have an observed string $\mathbf{w}$, the model defines a distribution over possible corrections:

$$\begin{aligned} P(\mathbf{t} \mid \mathbf{w}) &= \frac{P(\mathbf{w} \mid \mathbf{t})P(\mathbf{t})}{P(\mathbf{w})} \\ &= \frac{P(\mathbf{w} \mid \mathbf{t})P(\mathbf{t})}{\sum_{\mathbf{t}'} P(\mathbf{w} \mid \mathbf{t}')P(\mathbf{t}')}. \end{aligned} \tag{6.6}$$

In hard EM, we just took the highest-probability $\mathbf{t}$ from this distribution and pretended that we saw that as training data. In true EM, we consider *every possible* $\mathbf{t}$ and pretend that we saw it $P(\mathbf{t} \mid \mathbf{w})$ times. The idea of a fractional count is intuitively somewhat strange, but mathematically not a problem: we estimate probabilities by counting and dividing, and that works just as well when the counts are fractional.

The rest of this section describes how to do this. As a preview, when we use true EM, our two examples from above become:

```
observed: bootcharr? i recall is habdy.
guess:    bootcharr, i recall is hably.
```

and

```
observed: Hello :)
guess:    Hello :)
```

Still not perfect, but better. A stronger language model would know that "hably" is not a word.

Slow version of EM

Here's some pseudocode, but it's inefficient and should not be implemented.

initialize the model
repeat
  for each observed string $\mathbf{w}^{(i)}$ **do** ▷ E step
    for every possible correction $\mathbf{t}$ **do** ▷ intractable!
      compute $P(\mathbf{t} \mid \mathbf{w}^{(i)})$
      for each position j **do**
        $c(\mathbf{t}_j, \mathbf{w}_j^{(i)}) \mathrel{+}= P(\mathbf{t} \mid \mathbf{w}^{(i)})$
      end for
    end for

```
10:     end for
11:     for all w, t do                                              ▷ M step
12:         p(w | t) = c(t,w) / Σ_w' c(t,w')
13:     end for
14: until converged
```

The loop at line 4 is over an exponential number of corrections **t**, which is not practical. Actualy, it's worse than that, because according to equation (6.6), to even compute a single $P(\mathbf{t} \mid \mathbf{w}^{(i)})$ requires that we divide by $P(\mathbf{w}^{(i)})$, and this requires a sum over all $\mathbf{t}'$.

Forward algorithm

Let's tackle this problem first: how do we compute $P(\mathbf{w})$ efficiently? Just as the Viterbi algorithm uses dynamic programming to maximize over all $\mathbf{t}'$ efficiently, there's an efficient solution to sum over all $\mathbf{t}'$. It's a tiny modification to the Viterbi algorithm. We assume, as before, that we have a finite automaton that compactly represents all possible **t** (e.g., Figure 6.1), and the weight of a path is the probability $P(\mathbf{t}, \mathbf{w})$.

```
forward[q_0] ← 1
forward[q] ← 0 for q ≠ q_0
for each state q' in topological order do
    for each incoming transition q → q' with weight p do
        forward[q'] += forward[q] × p
    end for
end for
```

This is called the *forward algorithm*. The only difference is that wherever the Viterbi algorithm took a max of two weights, the forward algorithm adds them. Assuming that the automaton has a single final state q_f, we have forward$[q_f] = P(\mathbf{w})$, the total probability of all the paths in the automaton.

Fast version of EM

Now let's return to our pseudocode for EM. How do we do the loop over **t** efficiently? We rewrite the pseudocode in terms of the finite automaton and reorder the loops so that, instead of looping over all positions of all **t**, we now loop over the transitions of this automaton:

```
 1: initialize the model
 2: repeat
 3:     for each observed string w^(i) do                             ▷ E step
 4:         form automaton that generates all possible t with weight P(t, w^(i))
 5:         compute P(w^(i))                                          ▷ forward algorithm
 6:         for every transition e = (q_{i,t} → q_{i+1,t'}) do
 7:             compute weight p of all paths using transition e
 8:             c(t', w_{i+1}) += p / P(w^(i))                        ▷ because e corrects w_{i+1} to t'
 9:         end for
10:     end for
11:     for all w, t do                                               ▷ M step
12:         p(w | t) = c(t,w) / Σ_w' c(t,w')
```

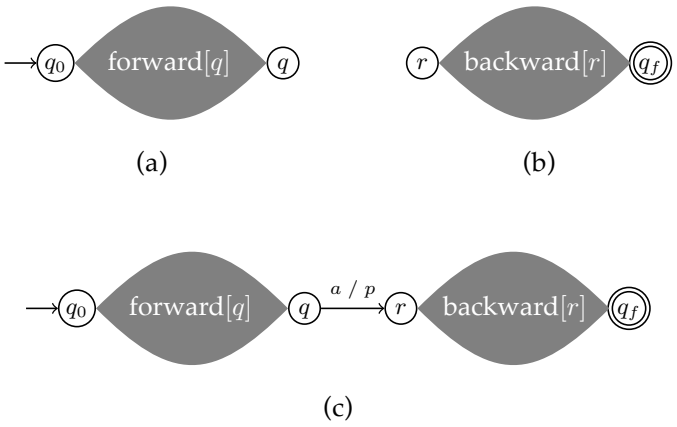


Figure 6.3: (a) The forward probability $\alpha[q]$ is the total weight of all paths from the start state to q. (b) The backward probability $\beta[q]$ is the total weight of all paths from to q to the final state. (c) The total weight of all paths going through an edge $q \rightarrow r$ with weight p is forward$[q] \times p \times$ backward$[r]$.

13: **end for**
14: **until** converged

Line 7 computes the total weight of all paths that go through transition e. To compute this, we need an additional helper function.

Backward algorithm

The *backward* weights are the mirror image of the forward weights: the backward weight of state q, written backward$[q]$, is the total weight of all paths from q to the final state q_f.

backward$[q_f] \leftarrow 1$
backward$[q] \leftarrow 0$ **for** $q \neq q_f$
for each state q in reverse topological order **do**
  for each outgoing transition $q \rightarrow q'$ with weight p **do**
    backward$[q]$ += $p \times$ backward$[q']$
  end for
end for

Note that forward$[q_f]$ = backward$[q_0]$; both are equal to $P(\mathbf{w}^{(i)})$, the total weight of all paths. When implementing the forward and backward algorithms, it's helpful to check this.

One more thing...

Now given transition $e = (q \xrightarrow{p} r)$, we want to use these quantities to compute the total weight of all paths using transition e. The total weight of all paths going into q is forward$[q]$, and the weight of e itself is p, and the total weight of all paths going out of r is backward$[r]$, so by multiplying all three of these together, we get the total weight of all paths going through e. See Figure 6.3.

So, we can implement line 7 above as

$$c(t', w_{i+1}) \mathrel{+}= \frac{\text{forward}[q] \times p \times \text{backward}[r]}{\text{forward}[q_f]}. \qquad e = (q \xrightarrow{p} r) \tag{6.7}$$

That completes the algorithm.

The remarkable thing about EM is that each iteration is guaranteed to improve the model, in the following sense. We measure how well the model fits the training data by the log-likelihood,

$$L = \sum_i \log P(\mathbf{w}^{(i)}).$$

In other words, a model is better if it assigns more probability to the strings we observed (and less probability to the strings we didn't observe). Each iteration of EM is guaranteed either to increase L, or else the model was already at a *local* maximum of L. Unfortunately, there's no guarantee that we will find a global maximum of L, so in practice we sometimes run EM multiple times from different random initial points.

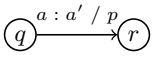
6.4 Finite-State Transducers

HMMs are used for a huge number of problems, not just in NLP and speech but also in computational biology and other fields. But they can get tricky to think about if the dependencies we want to model get complicated. For example, what if we want to use a trigram language model instead of a bigram language model? It can definitely be done with an HMM, but it might not be obvious how. If we want a text input method to be able to correct inserted or deleted characters, we really need something more flexible than an HMM.

In the last chapter, we saw how weighted finite-state automata provide a flexible way to define probability models over strings. But here, we need to do more than just assign probabilities to strings; we need to be able to transform them into other strings. To do that, we need *finite-state transducers*.

6.4.1 Definition

A finite-state transducer is like a finite-state automaton, but has both an input alphabet Σ and an output alphabet Σ'. The transitions look like this:



where $a \in \Sigma \cup \{\epsilon\}$, $a' \in \Sigma' \cup \{\epsilon\}$, and p is the weight.

A transition $a : \epsilon$ means "delete input symbol a," and $\epsilon : a'$ means "insert output symbol a'."

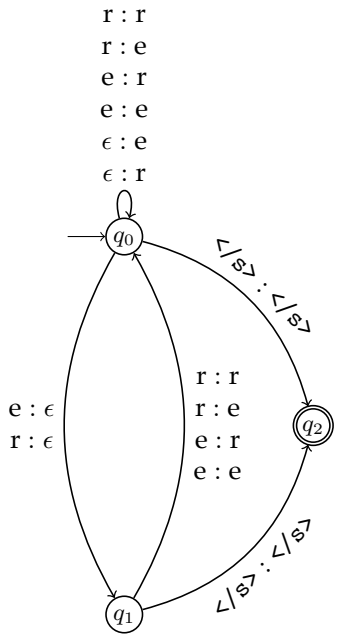
Whereas a FSA defines a set of strings, a FST defines a relation on strings (that is, a set of pairs of strings). A string pair $\langle w, w' \rangle$ belongs to this relation if there is a sequence of states $q_0, \ldots, q_n$ such that for all i, there is a transition $q_{i-1} \xrightarrow{w_i : w'_i} q_i$.

For now, the FSTs we're considering are deterministic in the sense that given an input/output string pair, there's at most one way for the FST to accept it. But given just an input string, there can be more than one output string that the FST can generate.

A *weighted FST* adds a weight to each transition. The weight of an accepting path through a FST is the product of the weights of the transitions along the path.

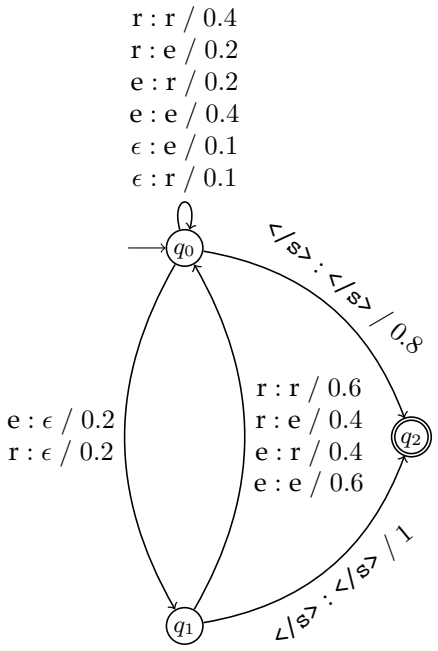
A *probabilistic FST* further has the property that for each state q *and* input symbol a, the weights of all of the transitions leaving q with input symbol a or ϵ sum to one. Then the weights of all the accepting paths *for a given input string* sum to one. That is, the WFST defines a conditional probability distribution $P(w' \mid w)$.

So a probabilistic FST is a way to define $P(\mathbf{w} \mid \mathbf{t})$. Let's consider a more sophisticated typo model that allows insertions and deletions. For simplicity, we'll restrict the alphabet to just the letters e and r. The typo model now looks like this:



Note that this allows at most one deletion at each position: after a deletion, we move to state q_1, in which no further deletions are allowed. The reason for this is that unlimited deletions will cause trouble for us later on.

Here's the same FST again, but with probabilities shown. Notice which transition weights sum to one.



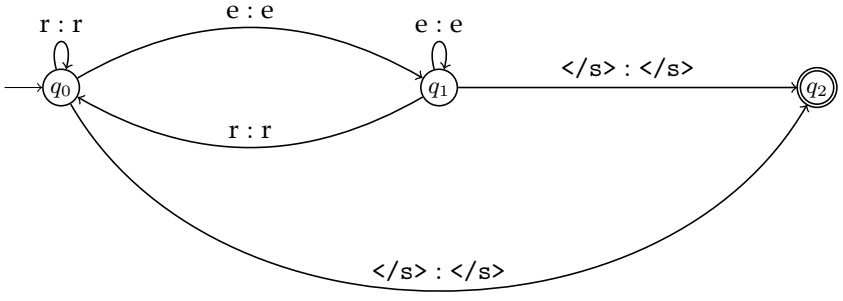
(6.8)

For example, for state q_0 and symbol e, we have

$$\begin{array}{ll} q_0 \xrightarrow{\text{e:r}} q_0 & 0.2 \\ q_0 \xrightarrow{\text{e:e}} q_0 & 0.4 \\ q_0 \xrightarrow{\epsilon\text{:e}} q_0 & 0.1 \\ q_0 \xrightarrow{\epsilon\text{:r}} q_0 & 0.1 \\ q_0 \xrightarrow{\text{e:}\epsilon} q_1 & 0.2 \\ \hline & 1 \end{array}$$

6.4.2 Composition

We have a probabilistic FSA, call it M_1, that generates sequences of true characters (a bigram model):



(6.9)

And we have a probabilistic FST, call it M_2, that models how the user might make typographical errors, shown above (6.8). Now, we'd like to combine them into a single machine that models both at the same time.

We're going to do this using FST *composition*. First, change M_1 into a FST that just copies its input to its output. We do this by replacing every transition

$$q \xrightarrow{a/p} r$$

with

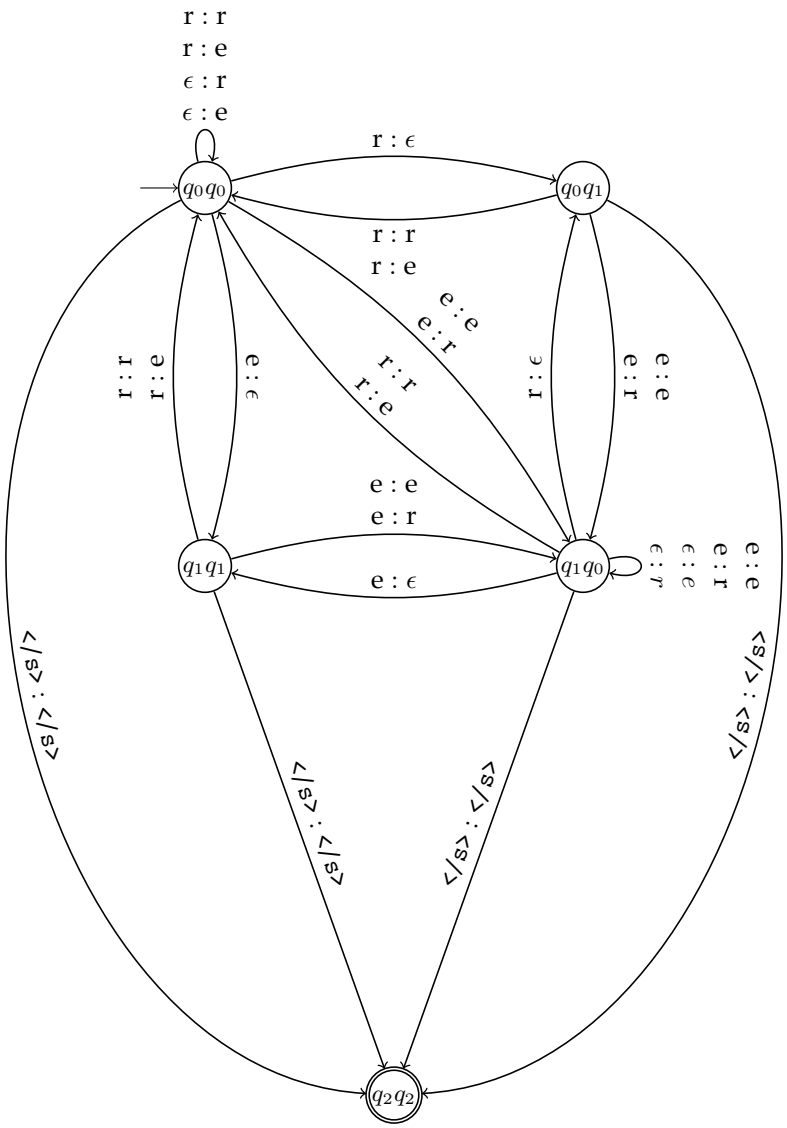
$$q \xrightarrow{a:a/p} r.$$

Then, we want to feed its output to the input of M_2. In general, we want to take any two FSTs M_1 and M_2 and make a new FST M that is equivalent to feeding the output of M_1 to the input of M_2 – this is the composition of M_1 and M_2. More formally, we want a FST M that accepts the relation $\{\langle u, w\rangle \mid \exists v \text{ s.t. } \langle u, v\rangle \in L(M_1), \langle v, w\rangle \in L(M_2)\}$.

If you are familiar with intersection of FSAs, this construction is quite similar. The states of M are pairs of states from M_1 and M_2. For brevity, we write state $\langle q_1, q_2\rangle$ as q_1q_2. The start state is s_1s_2, and the final states are $F_1 \times F_2$. Then, for each transition $q_1 \xrightarrow{a:b} r_1$ in M_1 and $q_2 \xrightarrow{b:c} r_2$ in M_2, make a new transition $q_1q_2 \xrightarrow{a:c} r_1r_2$. If the two old transitions have weights, the new transition gets the product of their weights.

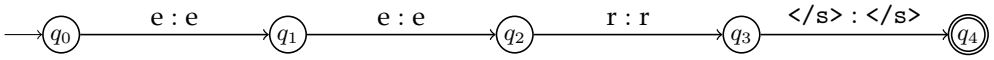
If we create duplicate transitions, you can either merge them while summing their weights, or sometimes it's more convenient to just leave them as duplicates.

For example, the composition of our bigram language model and typo model is:



(6.10)

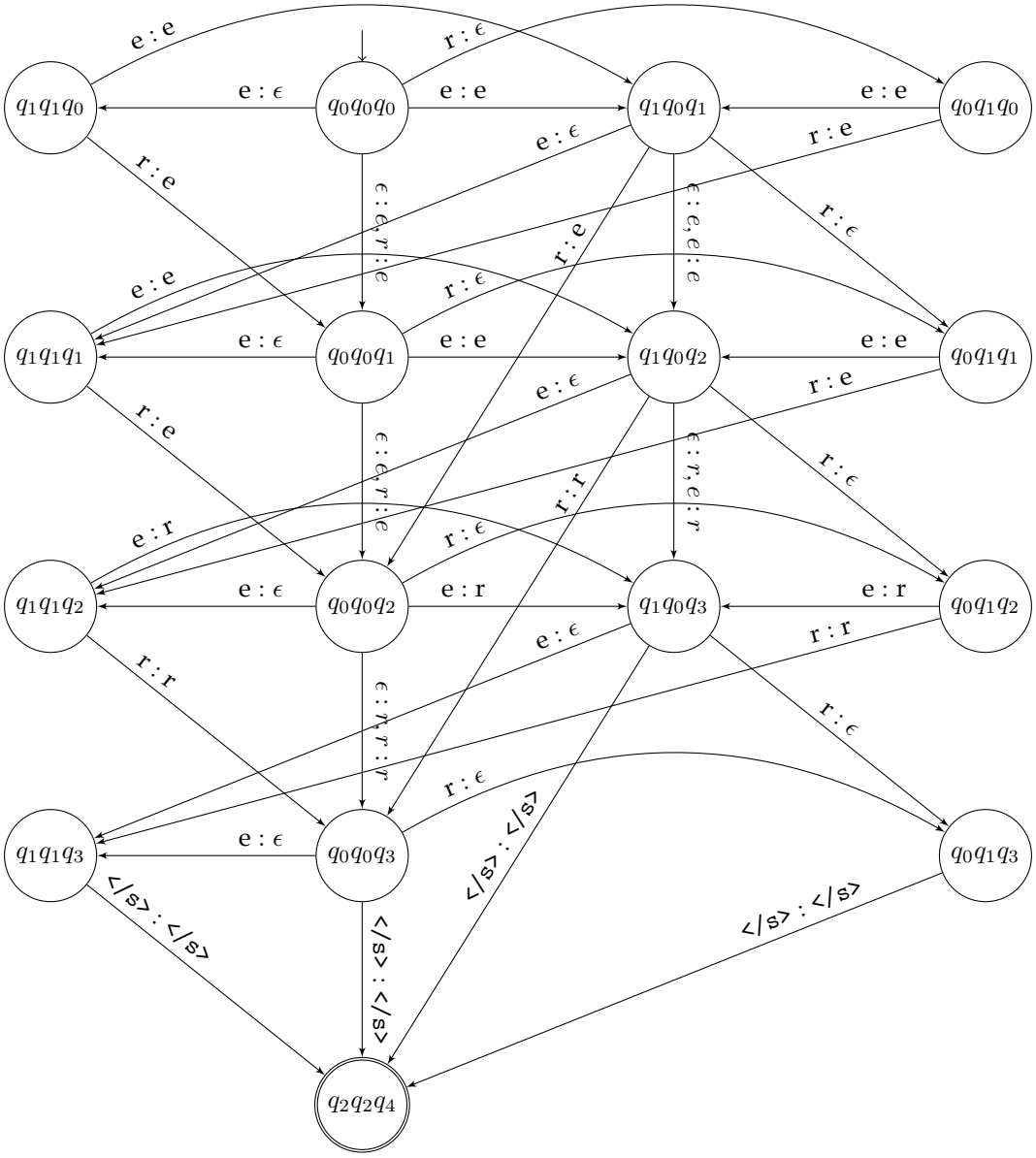
This is our typo-correction model in the form of a FST. Now, how do we use it? Given a string w of observed characters, construct an FST M_w that accepts only the string w as input and outputs the same string.



(6.11)

Now, if we compose the model with M_w, we get a new FST that accepts as input all possible true character sequences that could have given rise to w, and outputs only the string w. That FST looks

like this:



(6.12)

That looks very scary, but the good news is that the composition operation can be done completely automatically by a computer, so you will never have to construct one of these by hand, unless you teach a course in natural language processing. One important thing to notice about this transducer is that it is acyclic, and therefore there is only a finite (but large) number of possible true strings. This is *not* guaranteed; if we had allowed unlimited deletions, then for any observed

string w, there would be an infinite number of possible true strings. For example, if $w =$ ee and we allow just one deletion at each position, then eee, ree, ere, eer would be possible values of t, but if we allow unlimited deletions, then eeee, eeeee, eeeee, eeeeee, ... would also be possibilities.

But because the transducer is acyclic, we can run the Viterbi algorithm on it unmodified. This will give us the path with the highest $p(\mathbf{w} \mid \mathbf{t})$, and by looking at just the input labels of that path, we recover the best possible sequence of true characters $\mathbf{t}$.