Chapter 6

Sequence Labeling

6.1 Problem

Let's turn to a new task: given a sequence of words, we want to find the best way to assign a label to each of the words.

Part-of-speech tagging Given a sentence, assign a part-of-speech (POS) tag (noun, verb, etc.) to each word. Many words have a unique POS tag, but some words are ambiguous: for example, *short* can be an adjective (*short vowel*), a noun (direct a *short*), an adverb (to throw a ball *short*) or a verb (to *short* an appliance). Figuring out which POS is the correct one depends on the context, including the POS tags of the neighboring words.

Word sense disambiguation Given a dictionary which gives one or more *word senses* to each word (for example, a *bank* can either be a financial institution or the sloped ground next to a river), and given a sentence, guess the sense of each word in the sentence.

Named entity detection Given a sentence, identify all the proper names (Notre Dame, Apple, etc.) and classify them as persons, organizations, places, etc. The typical way to set this up as a sequence-labeling problem is called *BIO tagging*. Each word is labeled *B* (beginning) if it is the first word in a named entity, *I* (inside) if it is a subsequent word in a named entity, and *O* (outside) otherwise. Other encodings are possible as well.

Word segmentation Given a representation of a sentence without any word boundaries, reconstruct the word boundaries. In some languages, like Chinese, words are written without any spaces in between them. (Indeed, it can be difficult to settle on the definition of a "word" in such languages.) This situation also occurs with any spoken language.

6.2 Hidden Markov Models

6.2.1 Definition

As with the naïve Bayes classifier, instead of thinking directly about finding the most probable tagging of the sequence, we think about how the sequence might have come to be. Let $\mathbf{w} = w_1 \cdots w_n$ be a sequence of words and let $\mathbf{t} = t_1 \cdots t_n$ be a sequence of tags.

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

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

The first term, $P(\mathbf{t})$, can be described using a weighted FSA, just like a language model except over tags instead of words. 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 is even easier:

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

This is a *hidden Markov model* (HMM). If we're given labeled data, like

I	saw	her	duck
PRP	VBD	PRP$	NN

then the HMM is easy to train. But what we don't know how to do is classify (or *decode*): given **w**, what's the most probable **t**? Below, we'll see how to do that, 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.

Now, given the sentence "I saw her duck," we can construct a FSA that generates all possible POS tag sequences for this sentence. We'll just improvise it for now, and then see how to construct it methodically later. See Figure 6.1.

Note that this FSA is acyclic; a single run can never visit the same state twice, so it accepts strings of bounded length. Indeed, all the strings it accepts are exactly four symbols long. Given an acyclic FSA M, 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.

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:

$P(t' \mid t)$:

t' \ t	<s>	NN	PRP	PRP$	VB/VBD/VBP
NN	0	0.1	0	1	0.1
PRP	0.8	0	0	0	0.6
PRP$	0.2	0	0	0	0.2
VB	0	0.1	0.2	0	0
VBD	0	0.5	0.3	0	0
VBP	0	0	0.3	0	0
</s>	0	0.3	0.2	0	0.1

$P(w \mid t)$:

$p(\text{I} \mid \text{PRP}) = 0.5$
$p(\text{her} \mid \text{PRP}) = 0.5$
$p(\text{her} \mid \text{PRP\$}) = 1$
$p(\text{saw} \mid \text{VBD}) = 1$
$p(\text{saw} \mid \text{VBP}) = 1$
$p(\text{duck} \mid \text{NN}) = 1$
$p(\text{duck} \mid \text{VB}) = 1$

Table 6.1: Example parameters of an HMM.

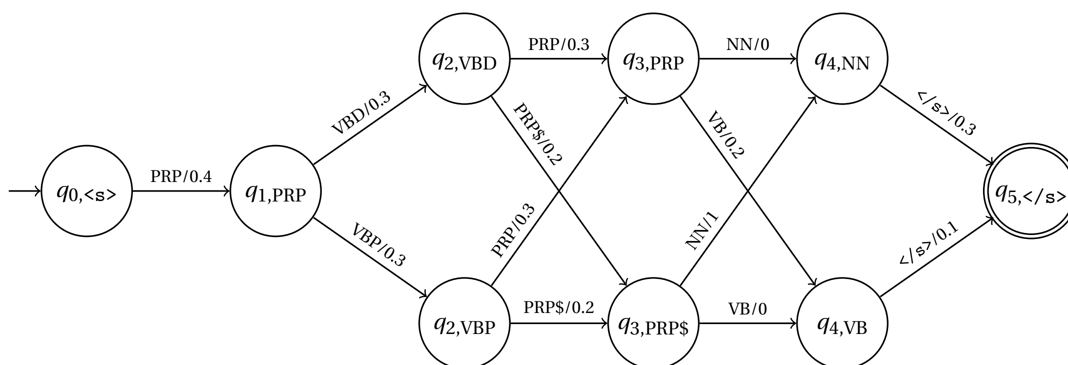


Figure 6.1: Weighted finite automaton for all taggings of sentence “I saw her duck.”

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

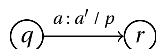
6.3 Finite-State Transducers

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, in the Chinese word segmentation problem, it would be terrible to use an HMM with just two tags B (beginning of word) and I (inside of word). It's critical for a tag to depend on the previous words, not just the previous tag. You can do it by modifying the tag set, but maybe we want a more flexible solution.

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.*

6.3.1 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 $a \in \Sigma$, $a' \in \Sigma'$, and p is the weight.

We start with a restricted definition; a more general definition will come later.

Definition 2. A *finite state transducer (FST)* is a tuple $\langle Q, \Sigma, \Sigma', \delta, s, F \rangle$, where:

- Q is a finite set of *states*
- Σ and Σ' are finite alphabets
- δ is a set of *transitions* of the form $q \xrightarrow{a:a'} r$, where
  - $q, r \in Q$
  - $a \in \Sigma$
  - $a' \in \Sigma'$

- $s \in Q$ is the *start state*
- $F \subseteq Q$ is the set of *final states*

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 $q_{i-1} \xrightarrow{w_i:w'_i} q_i$. The above definition defines FSTs that 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 is in general more than one output string that the FST can generate.

A *weighted FST* adds a weight to each transition. (Formally, we can redefine δ as a mapping $Q \times \Sigma \times \Sigma' \times Q \to \mathbb{R}$.) 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 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 $P(w' \mid w)$.

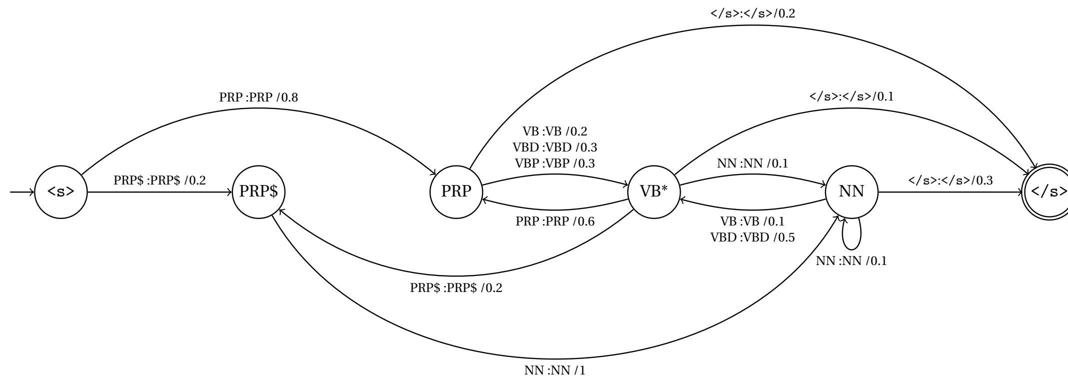
So a probabilistic FST is a way to define $P(\mathbf{w} \mid \mathbf{t})$. The simplest FST only has one state and many transitions. We'll write an example as a list of transitions instead of as a graph:

$$q \xrightarrow{\text{PRP:I/0.5}} q \qquad q \xrightarrow{\text{PRP:her/0.5}} q$$
$$q \xrightarrow{\text{PRP\$:her/1}} q$$
$$q \xrightarrow{\text{VBD:saw/1}} q$$
$$q \xrightarrow{\text{VBP:saw/1}} q$$
$$q \xrightarrow{\text{NN:duck/1}} q$$
$$q \xrightarrow{\text{VB:duck/1}} q$$
$$q \xrightarrow{\texttt{</s>:</s>/1}} q$$

Another very simple FST that we'll make use of is the *identity FST* corresponding to an FSA M, which just maps every string in $L(M)$ to itself. We do this by replacing every transition $q \xrightarrow{a} r$ with $q \xrightarrow{a:a} r$. For example, the bigram model $P(t' \mid t)$ that we saw above can be written as a FSA, and that FSA can be turned into its identity FST, which looks like this:

$$q_{\texttt{<s>}} \xrightarrow{\text{PRP:PRP/0.8}} q_{\text{PRP}} \qquad q_{\texttt{<s>}} \xrightarrow{\text{PRP\$:PRP\$/0.2}} q_{\text{PRP\$}}$$
$$q_{\text{NN}} \xrightarrow{\text{NN:NN/0.1}} q_{\text{NN}} \qquad q_{\text{NN}} \xrightarrow{\text{VB:VB/0.1}} q_{\text{VB*}} \qquad q_{\text{NN}} \xrightarrow{\text{VBD:VBD/0.5}} q_{\text{VB*}} \qquad q_{\text{NN}} \xrightarrow{\texttt{</s>:</s>/0.3}} q_{\texttt{</s>}}$$
$$q_{\text{PRP}} \xrightarrow{\text{VB:VB/0.2}} q_{\text{VB*}} \qquad q_{\text{PRP}} \xrightarrow{\text{VBD:VBD/0.3}} q_{\text{VB*}}$$
$$q_{\text{PRP}} \xrightarrow{\text{VBP:VBP/0.3}} q_{\text{VB*}} \qquad q_{\text{PRP}} \xrightarrow{\texttt{</s>:</s>/0.2}} q_{\texttt{</s>}}$$
$$q_{\text{PRP\$}} \xrightarrow{\text{NN:NN/1}} q_{\text{NN}}$$
$$q_{\text{VB*}} \xrightarrow{\text{NN:NN/0.1}} q_{\text{NN}} \qquad q_{\text{VB*}} \xrightarrow{\text{PRP:PRP/0.6}} q_{\text{PRP}}$$
$$q_{\text{VB*}} \xrightarrow{\text{PRP\$:PRP\$/0.2}} q_{\text{PRP\$}} \qquad q_{\text{VB*}} \xrightarrow{\texttt{</s>:</s>/0.1}} q_{\texttt{</s>}}$$

where we've written VB* as shorthand for any of VBD, VBP, or VB. All other probabilities should be assumed to be zero. As a graph, it looks like this:



6.3.2 Composition

We have a probabilistic FSA, call it M_1, that generates sequences of POS tags (a bigram model). And we have a probabilistic FST, call it M_2 that changes POS tags into words. Now, we'd like to combine them into a single machine that generates words together with their POS tags.

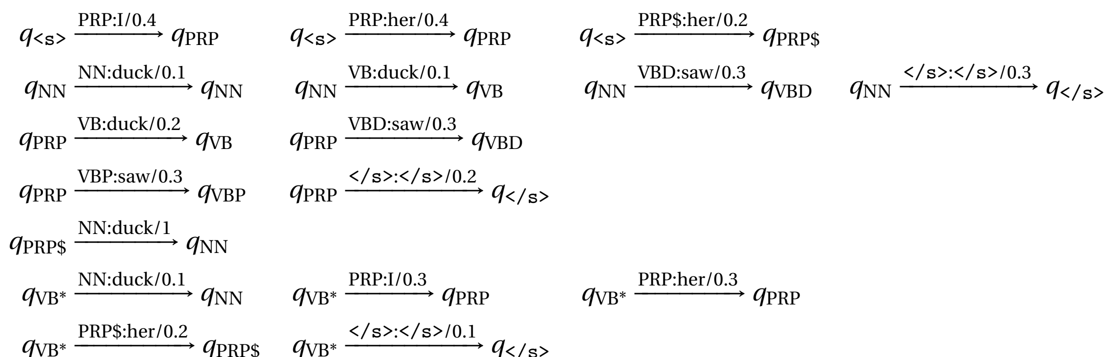
We're going to do this using FST *composition*. First, change M_1 into its identity FST: it reads in a POS tag sequence and writes out the exact same sequence. Then, we want to feed its output to the input of M_2. In general, we want to take any two FSTs M_1 and M_2 and make a new FST M that is equivalent to feeding the output of M_1 to the input of M_2 – this is the composition of M_1 and M_2.

If you are familiar with intersection of FSAs, this construction is quite similar. Given FSTs M_1 and M_2, we want a FST M that accepts the relation $\{\langle u, w\rangle \mid \exists v \text{ s.t. } \langle u, v\rangle \in L(M_1), \langle v, w\rangle \in L(M_2)\}$.

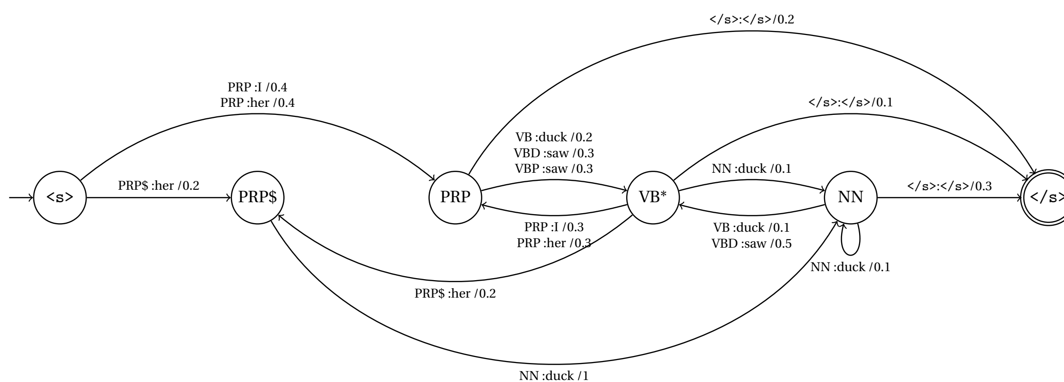
The construction is as follows: The states of M are pairs of states from M_1 and M_2. For brevity, we write state $\langle q_1, q_2\rangle$ as $q_1 q_2$. The start state is $s_1 s_2$, and the final states are $F_1 \times F_2$. Then, for each transition $q_1 \xrightarrow{a:b} r_1$ in M_1 and $q_2 \xrightarrow{b:c} r_2$ in M_2, make a new transition $q_1 q_2 \xrightarrow{a:c} r_1 r_2$. If the two old transitions have weights, the new transition gets the product of their weights. If we create duplicate transitions, merge them and sum their weights.

For example, suppose that M_1 is the FST version of the bigram model $P(t' \mid t)$ from above, and M_2 is

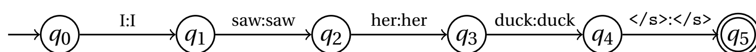
the one-state FST that we wrote for $P(w \mid t)$. Since M_2 has only one state, M has the same states as M_1.



As a graph:



One more step! Given a string w, construct an FST M_w that accepts only the string w as input and outputs the same string.



Now, if we compose M_w with the HMM, we get a new FST that outputs only the string w accepts as input all and only the possible tag sequences of w. Finally, if we ignore the output labels (the words) and look at the input labels (the tags), we have exactly the FSA that we constructed in Section 6.2.2.

6.4 Conditional random fields

Recall that we moved from naïve Bayes to logistic regression for a few reasons:

- To model $P(k \mid d)$ directly instead of modeling $P(k, d)$, which seems like more work.
- To allow adding arbitrary features without messing up the model.

We can make an analogous move from hidden Markov models to *conditional random fields* (Lafferty, McCallum, and Pereira, 2001). CRFs model $P(\mathbf{t} \mid \mathbf{w})$ directly instead of $P(\mathbf{t}, \mathbf{w})$, and they allow adding arbitrary features. They consistently outperform HMMs for sequence labeling tasks, and generally do as well or better than other methods. The downside of CRFs is that, like logistic regression, they have to be trained using numerical optimization, which is a lot slower than HMMs.

6.4.1 Definition

The model is defined as:

$$P(\mathbf{t} \mid \mathbf{w}) = \frac{\exp s(\mathbf{t}, \mathbf{w})}{\sum_{\bar{\mathbf{t}}} \exp s(\bar{\mathbf{t}}, \mathbf{w})} \tag{6.5}$$

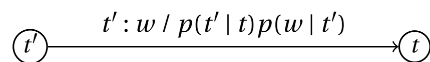
$$s(\mathbf{t}, \mathbf{w}) = \lambda(\texttt{<s>}, t_1) + \mu(t_1, w_1) + \left(\sum_{i=2}^{n} \lambda(t_{i-1}, t_i) + \mu(t_i, w_i) \right) + \lambda(t_n, \texttt{</s>}). \tag{6.6}$$

where $\bar{\mathbf{t}}$ ranges over all possible tag sequences of length n. (See below for how to compute this sum efficiently.)

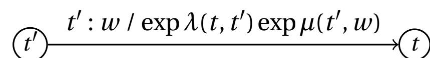
There are two kinds of weights: $\lambda(t, t')$ for every tag/tag pair, and $\mu(t, w)$ for every tag/word pair. As with logistic regression, there's no formula for the weights; you initialize them to all zeros and optimize them using (stochastic) gradient ascent.

6.4.2 As finite transducers

Recall that we wrote an HMM as a finite transducer, where the edges looked like



We can write a CRF as a finite transducer too:



If we run this transducer on tags $\mathbf{t}$ and words $\mathbf{w}$, then the weight of the path is $\exp s(\mathbf{t}, \mathbf{w})$, which is the numerator of (6.5). How do we compute the denominator?

6.4.3 Summing over all tag sequences

The denominator in equation (6.5) is a summation over all possible tag sequences $\bar{\mathbf{t}}$ – that's a lot of tag sequences! The way that we perform the summation is similar to how the Viterbi algorithm finds the maximum over all possible tag sequences. The only difference is that when a state has multiple incoming edges, we add the weights instead of taking their maximum.

forward$[q_0] \leftarrow 1$
forward$[q] \leftarrow 0$ **for** $q \neq q_0$
for each state q' in topological order **do**
  for each incoming transition $q \rightarrow q'$ with weight p **do**
    forward$[q'] \leftarrow$ forward$[q']$ + forward$[q] \times p$
  end for
end for

6.4.4 Training

As always, we want to maximize the log-likelihood:

$$\log L = \sum_{\mathbf{t}, \mathbf{w} \text{ in data}} \log P(\mathbf{t} \mid \mathbf{w}) \tag{6.7}$$

We do this using stochastic gradient descent, looping over all the sentences, and for each sentence $\mathbf{w}$ with correct tags $\mathbf{t}$, take a step uphill on $\log P(\mathbf{t} \mid \mathbf{w})$. The gradient can be computed by automatic differentiation.

Bibliography

Lafferty, John, Andrew McCallum, and Fernando Pereira (2001). "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data". In: *ICML*, pp. 282–289.