Lecture 8: Using HMM for Tagging Applications

Readings: Manning & Schutze, Section 7.3
Jurafsky and Martin, Chapter 9

Many problems can be cast as **tagging problems**. In a tagging problem, we need to associate a value (a **tag**) with each element in a corpus. We have already seen some simple techniques for part-of-speech tagging (see sample tag set in Figure 1) using probabilistic models based on the word. Tagging can be used for many purposes, such as the following:

1. We could identify word boundaries in language like Japanese where spaces are not used as we do in English. In this case, we'd be tagging each letter as to whether it begins a new word or not;
2. We could identify noun phrases in a corpus by labeling each word as to whether it begins an NP, is in an NP or ends an NP;
3. We could tag words as to whether they are part of a self-correction or not, and what role they play in the correction (e.g., I want a coffee ah a tea).

1. Probability Models for Tagging Problems

Consider the tagging problem in general: Let $W_{1,T} = w_1, ..., w_T$ be a sequence of T words (or letters or any other element in a corpus). We want to find the sequence of tags $C_{1,T} = c1, ..., c_T$ that maximizes the probability of the tag sequence given the word sequence, i.e.,

1. $\text{argmax}_{C_{1,T}} P(C_{1,T} \mid W_{1,T})$ (i.e., $P(c_1, ..., c_T \mid w_1, ..., w_T)$)

Unfortunately, it would take far too much data to generate reasonable estimates for such sequences, so direct methods cannot be applied. There are, however, reasonable approximation techniques that produce good results. To develop them, we restate formula (1) using Bayes' rule, which says that conditional probability (1) equals

2. $\text{argmax}_{C_{1,T}} (P(C_{1,T}) * P(W_{1,T} \mid C_{1,T})) / P(W_{1,T})$

Since we are interested in finding the $C_{1,T}$ that gives the maximum value, the common denominator in all these cases will not affect the answer. Thus the problem reduces to finding the sequence $C_{1,T}$ that maximizes the formula

$$\text{argmax}_{C_{1,T}} P(C_{1,T}) * P(W_{1,T} \mid C_{1,T})$$

There are still no effective methods for calculating the probability of these long sequences accurately, as it would still require far too much data! Note also that the techniques we have developed so far all depend on the Markov assumption of having a limited context, and we have not yet made this assumption in this problem as currently formulated. So we must approximate these probabilities using some independence assumptions that will limit the context. While these independence assumptions may not be really valid, the estimates appear to work reasonably well in practice. Each of the two expressions in formula 3 will be approximated. The first expression, the probability of the sequence of categories, can be approximated by a series of probabilities based on a limited number of previous categories. We can develop this idea by first noting that using the Chain Rule we know that $P(C_{1,T})$ is equal to

$$P(c_T \mid c_1, \ldots, c_{T-1}) * P(c_{T-1} \mid c_1, \ldots, c_{T-2}) * \ldots * P(c_1)$$

We then simplify this formula by assuming that an element only depends on only the n most recent categories. The **bigram** model looks at pairs of categories (or words) and

1.	CC	Coordinating conjunction	19.	PP$	Possessive pronoun
2.	CD	Cardinal number	20.	RB	Adverb
3.	DT	Determiner	21.	RBR	Comparative adverb
4.	EX	Existential *there*	22.	RBS	Superlative Adverb
5.	FW	Foreign word	23.	RP	Particle
6.	IN	Preposition / subord. conj	24.	SYM	Symbol (math or scientific)
7.	JJ	Adjective	25.	TO	to
8.	JJR	Comparative adjective	26.	UH	Interjection
9.	JJS	Superlative adjective	27.	VB	Verb, base form
10.	LS	List item marker	28.	VBD	Verb, past tense
11.	MD	Modal	29.	VBG	Verb, gerund/pres. participle
12.	NN	Noun, singular or mass	30.	VBN	Verb, past participle
13.	NNS	Noun, plural	31.	VBP	Verb, non-3s, present
14.	NNP	Proper noun, singular	32.	VBZ	Verb, 3s, present
15.	NNPS	Proper noun, plural	33.	WDT	Wh-determiner
16.	PDT	Predeterminer	34.	WP	Wh-pronoun
17.	POS	Possessive ending	35.	WPZ	Possessive wh-pronoun
18.	PRP	Personal pronoun	36.	WRB	Wh-adverb

Figure 1 The Penn Treebank tagset

uses the conditional probability that a category C_i will follow a category C_{i-1}, written as $P(C_i \mid C_{i-1})$. The **trigram** model uses the conditional probability of one category (or word) given the two preceding categories (or words), that is, $P(c_i \mid c_{i-2}\ c_{i-1})$. These models are called **n-gram** models, in which n represents the number of words used in the pattern. Using bigrams, the following approximation can be used:

$$P(C_{1,T}) \approx \prod_i P(c_i \mid c_{i-1})$$

To account for the beginning of a sentence, we posit a pseudo-category $\emptyset$ as the value of c_0. Thus the first bigram for a sentence beginning with an ART would be P(ART | ø). Given this, the approximation of the probability of the sequence ART N V N using bigrams would be

$$P(\text{ART N V N}) \approx P(\text{ART} \mid \emptyset) * P(\text{N} \mid \text{ART}) * P(\text{V} \mid \text{N}) * P(\text{N} \mid \text{V})$$

The second probability in formula 3, i.e.,

$$P(w_1, ..., w_T \mid c_1, ..., c_T)$$

can be approximated by limiting the context used for each w_i. In particular, using the chain rule again we know this formula is equal to

$$P(w_T \mid w_1, ..., w_{T-1,}\ c_1, ..., c_T) * P(w_{T-1} \mid w_1, ..., w_{T-2,}\ c_1, ..., c_T) * P(w_1 \mid c_1, ...,$$

Why do we always apply Bayes' Rule?

You will see that in almost all case, the first thing done in creating a viable probabilistic model is to apply Bayes Rule. Thus, to formulate a model of

(a) $\text{argmax}_{C_{1,T}} P(C_{1,T} \mid W_{1,T})$

we rewrite it as

(b) $\text{argmax}_{C_{1,T}} (P(C_{1,T}) * P(W_{1,T} \mid C_{1,T})) / P(W_{1,T})$.

Why is this? To explore this question, let's expand $P(C_{1,T} \mid W_{1,T})$ using the Chain rule:

(c) $P(c_T \mid W_{1,T}\ C_{1,T-1}) * P(c_{T-1} \mid W_{1,T}\ C_{1,T-2}) * \ldots * P(c_1 \mid W_{1,T})$

We now make independence assumptions to produce model we can reasonably estimate. The analog to the simple bigram model we develop in the text would have c_i independent of everything except c_{i-1} and w_i, and thus our model would be

(d) $\prod_i P(c_i \mid c_{i-1}\ w_i)$

Note the main difference here is that this model requires a joint distribution across three random variables, whereas the Bayes formulation breaks the problems into estimate two distributions with two random variables each. Furthermore, note that this model is conditioned on W_i, which has a very large vocabulary. This would make it hard to construct good estimates. In applications where there is sufficient data, however, there is no reason why models along these lines would not be just as effective. The other thing to note about this formulation is that there is also not a straightforward mapping to HMMs or the Noisy Channel Model. While not necessarily a flaw, it means that the development of a Viterbi style algorithm for searching this model would be a little harder to motivate.

Context	Count at i-1	Pair	Count at i-1,i	Bigram	Estimate
ø	300	ø, ART	213	*P*(ART | ø)	.71
ø		ø, N	87	*P*(N | ø)	.29
ART	558	ART, N	558	*P*(N | ART)	1
N	833	N, V	358	*P*(V | N)	.43
N		N, N	108	*P*(N | N)	.13
N		N, P	366	*P*(P | N)	.44
V	300	V, N	75	*P*(N | V)	.35
V		V, ART	194	*P*(ART | V)	.65
P	307	P, ART	226	*P*(ART | P)	.74
P		P, N	81	*P*(N | P)	.26

Figure 2 Bigram probabilities from the generated corpus

c_T)

Here we could make assumptions that combine say, n previous words and m previous categories. The simplest such model is to assume that a word appears in a category independent of whatever the preceding words and categories are. With this assumption, the formula would be approximated by the product of the probability that each word occurs in the indicated part of speech, that is, by

$$P(w_1, ..., w_T \mid c_1, ..., c_T) \approx \prod_{i=1,T} P(w_i \mid c_i)$$

With these two approximations, the problem has changed into finding the sequence such that

$$\text{argmax}_{C_{1,T}} \prod_{i=1,T} P(c_i \mid c_{i-1}) * P(w_i \mid c_i)$$

This in the classic **bigram model of tagging**. The probabilities involved can be readily estimated from a corpus of text labeled with parts of speech. In particular, given a database of text, the bigram probabilities can be estimated simply by counting the number of times each pair of categories occurs compared to the individual category counts. The probability that a V follows an N would be estimated as follows:

$$P(C_i=V \mid C_{i-1}=N) \approx \frac{\text{Count(N at position } i-1 \text{ and V at } i)}{\text{Count(N at position } i-1)}$$

Figure 2 gives some bigram frequencies computed from an artificially generated corpus of simple sentences. The corpus consists of 300 sentences but has words in only four categories: N, V, ART, and P. The artificial corpus contains 1998 words: 833 nouns, 300 verbs, 558 articles, and 307 prepositions. The first column is the tag at position i-1, and the second is how many times it occurs. The third column gives the bigrams, and the fourth the count of bigrams. From these two counts, the MLE probability estimate for the conditional probabilities in column 5 is given in column 6. To deal with the problem of sparse data, we could use one of the techniques from last time. For this example, however, let's just assume give any unseen event a token probability of .0001.

The lexical-generation probabilities, $P(w_i \mid C_i)$, can be estimated simply by counting the number of occurrences of each word by category. Figure 3 gives some counts for individual words from which the lexical-generation probability estimates in Figure 4 are computed. Note that the lexical-generation probability is the probability that a given category is realized by a specific word, not the probability that a given word falls in a specific category. For instance, P(*the* | ART) is estimated by Count (# times *the* is an ART) / Count (# times an ART occurs). The other probability, *P(*ART | *the*), would give a very different value.

	N	V	ART	P	TOTAL
flies	21	23	0	0	44
fruit	49	5	1	0	55
like	10	30	0	21	61
a	1	0	201	0	202
the	1	0	300	2	303
flower	53	15	0	0	68
flowers	42	16	0	0	58
birds	64	1	0	0	65
others	592	210	56	284	1142
TOTAL	833	300	558	307	1998

Figure 3 A summary of some of the word counts in the corpus

P(the | ART)	.54	*P(a* | ART)	.360
P(flies | N)	.025	*P(a* | N)	.001
P(flies | V)	.076	*P(flower* | N)	.063
P(like | V)	.1	*P(flower* | V)	.05
P(like | P)	.068	*P(birds* | N)	.076
P(like | N)	.012		

Figure 4 The lexical-generation probabilities

2. Finding the Maximum Probability Sequence

Once we have a probabilistic model, the next challenge is to find an effective algorithm for finding the maximum probability tag sequence given an input. It should be clear that an algorithm that enumerates every possible path will run in exponential time, for there are N^T possible path sequences of length T given N possible tags. Luckily we do not need to enumerate these paths because the nature of the models allows us to construct a **dynamic programming** algorithm, which for HMMs is called the **Viterbi Algorithm**.

This can be explored most intuitively by mapping the problem to an HMM in which the categories c_i become the states, the category bigrams become the transition probabilities, and $P(w_i \mid c_i)$ are the output probabilities. Figure 5 show the state transition probabilities as a finite state machine derived from the bigram statistics shown in Figure 2.

Given all these probability estimates, we can now return to the problem of finding the sequence of categories that has the highest probability of generating an observed sequence of outputs? This problem is related to the calculation of forward probabilities that we discussed earlier. (Remember, the forward probability Forward($S, o_1, \ldots, o_T$) was the probability that a machine would be in state S at time t after outputting each o_i at time i.) But this time we are looking for the single maximal probability path.

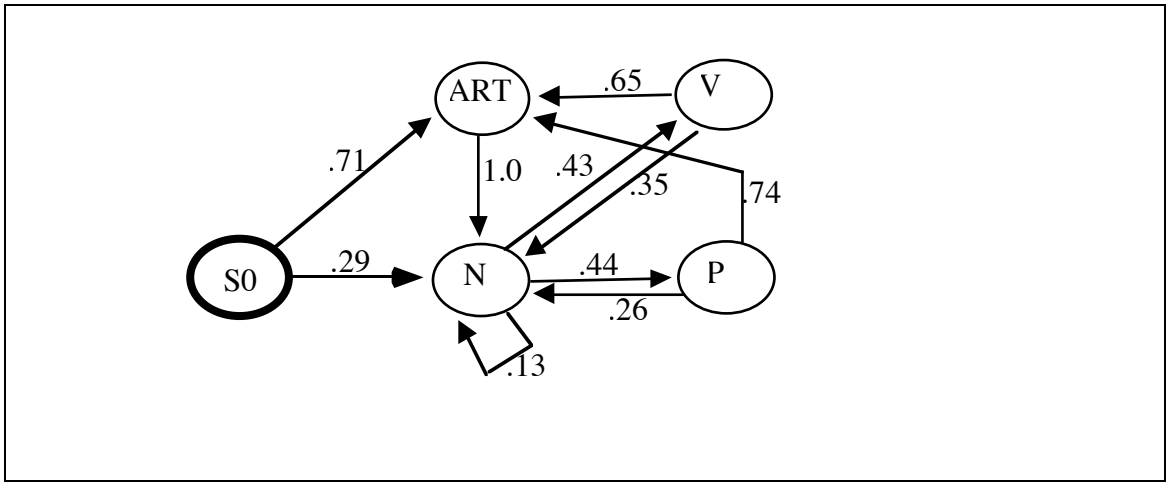

Figure 5 The HMM transitions capturing the bigram probabilities

Shortest Path Algorithms

In order to develop the core intuition about this algorithm, consider first a related algorithm that finds the shortest path between distances. Say we want to find the shortest path from Rochester (R) to New York City (NYC) using the map shown in Figure 6. We can find the best path by searching forward in a "breadth-first" fashion over the number of legs in the trip. So from R, we know we can get to M or C in one leg, and the distance to M is 30, and to C is 90. We record the shortest distance to a location and also, to remember how we got there, we record the city that we came from. Thus, we have the following state record after one leg:

M: 30 (R) "Manchester, 30 miles coming from Rochester)

C : 90 (R)

I: 110 (R)

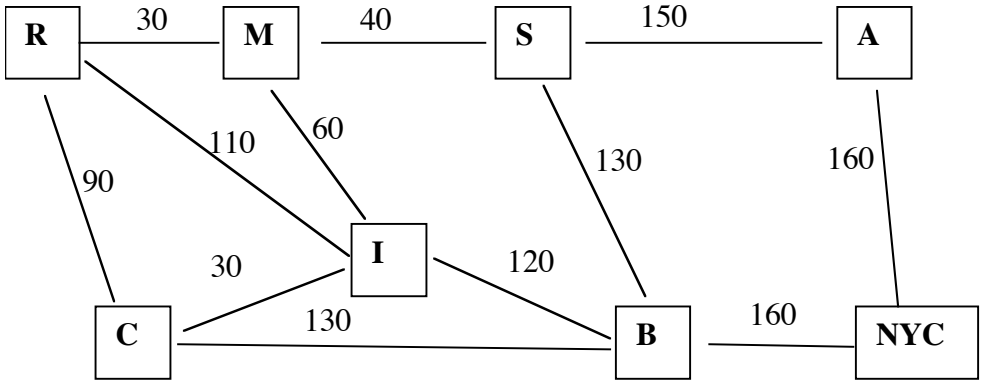

Figure 6: Getting from Rochester to New York City

Now we expand out another leg from each of these cities. From M we get to S in 70, and I in 90. Since this is better than the previous distance found (110 miles from R), we replace the old estimate with the new one. From C we get to B in 220, and to I in 120.

Since the distance to I is worse than our best estimate so far, we simply ignore it. From I we can get to B in 230 miles (the old route to I + 120). Thus is worse than the existing estimate so we ignore it as well. Thus we have the following

S: 70 (M)

I: 90 (M)

B: 220 (C)

Now we expand out the third leg. From S, we can get to B in 200. Since this is better than the previous route found to B via C, we update what is the best route for B.

When we expand out the legs from I, we also get to B, but this time the distance, 210, is not better than what we've already found, so we ignore it. The record after expanding all possible leg 3 will be:

B: 200 (S)

A: 220 (S)

The final round of expansions gets us to NYC from B with 360 miles, better than the 380 miles coming from Albany. Once we have complete the search, we can then find the best route but searching backwards through the record of previous cities. We got to NYC from B, to B from S, to S from M, and to M from R. Thus our route is R M S B NYC.

The Viterbi Algorithm

We now want to develop a similar algorithm for HMMs. Consider searching the above HMM on the input *flies like a flower*. We immediately see that there are a number of differences to deal with:

1. We are computing a maximum probability rather than a minimum distance. This is simple to change the formulas used in the algorithm.
2. With finding routes, we know where we are at each point in time. With an HMM we don't. In other words, seeing the first word *flies* doesn't tell us whether we are in state N, V, ART or P. We need to generalize the algorithm to deal with probability distributions over states.
3. With finding routes, we never had to visit the same state twice. With HMMS, we do. We will account for this but keeping a separate record for each state at each time point – thus state N at time 2 will have a different record than state N at time 1.
4. With HHMs, we don't necessarily have a known final state, we may be looking for the best path no matter what state it ends in.

But none of these problems modify the algorithm in any substantial way. We can view the algorithm as filling in a chart of the following form, with time running horizontally and the states vertically, where each value indicates the maximum probability path found so far to this state at this time, and a pointer to the previous state at time t-1 on the maximum path. We show the results from the first iteration, where we compute the probability of the transition to each state from the starting state ø, times the output probability of *flies* given the state.

	Flies	Like	A	flower
N	$7.25 * 10^{-3}$, ø			

V	$7.6 * 10^{-6}$, ø			
ART	$7.1 * 10^{-5}$, ø			
P	$1 * 10^{-8}$, ø			

Developing this more carefully, let MaxProb(S_i, o_1 … o_t) be the maximum probability of a path to S_i at time t that has output o_1 … o_t. We know that the output at time 1 is *flies*, and the maximum probability path to each state is simply the initial probability distribution, which we are modeling as the bigram conditional probability P(S_i | ø) times the output probability:

MaxProb(s_i, *flies*) = P(s_i | ø) * P(flies | s_i)

(*Which, written more precisely using random variables C_i for the state at time i and O_i for the output at time i is*

*MaxProb(s_i, flies) = P(C_1=s_i | C_0=ø) * P(O_1=flower | C_1=s_i))*

Using this we compute the following:

MaxProb(N, *flies*) = .29 * .025 = $7.25 * 10^{-3}$

MaxProb(V, *flies*) = .0001 * .076 = $7.6 * 10^{-6}$

MaxProb(ART, *flies*) = .71 * .0001 = $7.1 * 10^{-5}$

MaxProb(P, *flies*) = .0001 * .0001 = $1 * 10^{-8}$

We fill in the rest of the chart using an iterative process, where at each stage of the iteration, it uses the maximum probability of paths found to the states at time i to compute the maximum probability of paths for the states at time i+1.

We now can compute the probability for subsequent times using the formula

$$\text{MaxProb}(s_i, w_1 \ldots w_t) = \mathbf{Max}_j(\text{MaxProb}(s_j, w_1 \ldots w_{t-1}) * P(C_t=s_i \mid C_{t-1}=s_j)) * P(O_t=w_t \mid C_t=s_i))$$

i.e., the maximum path to s_i at time t will be the path that maximizes the maximum path to a state s_j at time t-1, times the transition from s_j to s_i times the output probability of w_t from state s_i.

Consider using this formula to compute MaxProb(V, *flies like)*, we take the maximum of four possible values:

Maximum

{MaxProb(ART, *flies*) * P(V | ART), MaxProb(N, *flies*) * P(V | N),
MaxProb(V, *flies*) * P(V | V), MaxProb(P, *flies*) * P(V | P)} * P(*like* | V)}

The maximum of these will be the path that comes from N, which is

MaxProb(V, *flies like*) = .00975 * .43 * .1 = .00042.

We would likewise compute the probabilities of the maximum paths to states N, P and ADJ, and then repeat the process for time step 3. As before, the amount of computation required at each time step is constant, and proportional to N^2, where N is the number of states**.**

How Good are the Basic Tagging Models?

Tagging algorithms using the Viterbi algorithm can perform effectively if the probability estimates are computed from a large corpus of data that is of the same style as the input to be classified. Researchers consistently report part-of-speech tagging accuracy for a well-trained bigram model of around 95 percent or better accuracy using trigram models. Remember, however, that the naive algorithm picks the most likely category about 90 percent of the time.

The Viterbi Algorithm in Detail

We will track the probability of the best sequence leading to each possible category at each position using an N¥T array, where N is the number states and T is the number of words in the sequence. This array, s_t (i), records the probability for the best sequence up to position t that ends in state i at time t. To record the actual best sequence for each category at each position, it suffices to record only the one preceding category for each category and position. Another N¥T array, BACKPTR, will indicate for each state in each position what the preceding state produced sequence at position t-1

Given word sequence w_1, ..., w_T, output types L_1, ..., L_N, output probabilities $P(w_t \mid L_i)$, and transition probabilities $P(L_i \mid L_j)$, find the most likely sequence of states S_1, ..., S_T for the output sequence.

Initialization Step

For i = 1 to N do

$\delta_1(i) = P(w_1 \mid L_i) * P(L_i \mid \emptyset)$

BACKPTR(i, 1) = 0

Iteration Step

For t = 2 to T

For i = 1 to N

$\delta_t(i) = MAX_{j=1,N}(SEQSCORE(j, t-1) * P(L_i \mid L_j)) * P(w_t \mid L_i)$

BACKPTR(i, t) = index of j that gave the max above

Sequence Identification Step:

S(T) = i that maximizes $\delta_T(i)$

For i = T–1 to 1 do

S(i) = BACKPTR(PATH(i+1), i+1)