Part-of-Speech Tagging

Informatics 2A: Lecture 17

Adam Lopez

School of Informatics
University of Edinburgh

27 October 2016

Last class

We discussed the POS tag lexicon

When do words belong to the same class? Three criteria

What tagset should we use?

What are the sources of ambiguity for POS tagging?

1. Methods for tagging
   - Unigram tagging
   - Bigram tagging
   - Tagging using Hidden Markov Models: Viterbi algorithm

Reading: Jurafsky & Martin, chapters (5 and) 6.

Before we get started

Question you should always ask yourself:

How hard is this problem?

Before we get started

Question you should always ask yourself:

How hard is this problem?

For POS tagging, this boils down to:

How ambiguous are parts of speech, really?

If most words have unambiguous POS, then we can probably write a simple program that solves POS tagging with just a lookup table. E.g. "Whenever I see the word *the*, output DT."

Before we get started

Question you should always ask yourself:

How hard is this problem?

For POS tagging, this boils down to:

How ambiguous are parts of speech, really?

If most words have unambiguous POS, then we can probably write a simple program that solves POS tagging with just a lookup table. E.g. "Whenever I see the word *the*, output DT."

This is an empirical question. To answer it, we need data.

Corpus annotation

A corpus (plural corpora) is a computer-readable collection of natural language text (or speech) used as a source of information about the language: e.g. what words or constructions can occur in practice, and with what frequencies?
To answer the question about POS ambiguity, we use corpora in which each word has been *annotated* with its POS tag, e.g.

```
Our/PRP\$ enemies/NNS are/VBP innovative/JJ and/CC
resourceful/JJ ,/, and/CC so/RB are/VB we/PRP ./.
They/PRP never/RB stop/VB thinking/VBG about/IN new/JJ
ways/NNS to/TO harm/VB our/PRP\$ country/NN and/CC
our/PRP\$ people/NN, and/CC neither/DT do/VB we/PRP ./.
```

Empirical upper bounds: inter-annotator agreement

Even for humans, tagging sometimes poses difficult decisions.
E.g. Words in -ing: adjectives (JJ), or verbs in gerund form (VBG)?

a boring/JJ lecture	a *very* boring lecture
	? a lecture that bores
the falling/VBG leaves	*the *very* falling leaves
	the leaves that fall
a revolving/VBG? door	*a *very* revolving door
	a door that revolves
	*the door seems revolving
sparkling/JJ? lemonade	? *very* sparkling lemonade
	lemonade that sparkles
	the lemonade seems sparkling

In view of such problems, we can't expect 100% accuracy from an automatic tagger.
In the Penn Treebank, annotators disagree around 3.5% of the time. Put another way: if we assume that one annotator tags perfectly, and then measure the accuracy of another annotator by comparing with the first, they will only be right about 96.5% of the time. We can hardly expect a machine to do better!

Word types and tokens

- Need to distinguish word tokens (particular occurrences in a text) from word types (distinct vocabulary items).
- We'll count different inflected or derived forms (e.g. break, breaks, breaking) as distinct word types.
- A single word type (e.g. **still**) may appear with several POS.
- But most words have a clear most frequent POS.

Question: How many tokens and types in the following? Ignore case and punctuation.

> *Esau sawed wood. Esau Wood would saw wood. Oh, the wood Wood would saw!*

1. 14 tokens, 6 types
2. 14 tokens, 7 types
3. 14 tokens, 8 types
4. None of the above.

Extent of POS Ambiguity

The Brown corpus (1,000,000 word tokens) has 39,440 different word types.

- 35340 have only 1 POS tag anywhere in corpus (89.6%)
- 4100 (10.4%) have 2 to 7 POS tags

So why does just 10.4% POS-tag ambiguity by word type lead to difficulty?
This is thanks to *Zipfian distribution*: many high-frequency words have more than one POS tag.
In fact, more than 40% of the word tokens are ambiguous.

He wants **to/TO** go.
He went **to/IN** the store.

He wants **that/DT** hat.
It is obvious **that/CS** he wants a hat.
He wants a hat **that/WPS** fits.

Word Frequencies in Different Languages

Ambiguity by part-of-speech tags:

Language	**Type-ambiguous**	**Token-ambiguous**
English	13.2%	56.2%
Greek	<1%	19.14%
Japanese	7.6%	50.2%
Czech	<1%	14.5%
Turkish	2.5%	35.2%

Word Frequency – Properties of Words in Use

Take any corpus of English like the Brown Corpus or Tom Sawyer and sort its words by how often they occur.

word	Freq. (f)	Rank (r)	$f \cdot r$
the	3332	1	3332
and	2972	2	5944
a	1775	3	5235
he	877	10	8770
but	410	20	8400
be	294	30	8820
there	222	40	8880
one	172	50	8600
about	158	60	9480
more	138	70	9660
never	124	80	9920
Oh	116	90	10440

Word Frequency – Properties of Words in Use

Take any corpus of English like the Brown Corpus or Tom Sawyer and sort its words by how often they occur.

word	Freq. (f)	Rank (r)	$f \cdot r$
two	104	100	10400
turned	51	200	10200
you'll	30	300	9000
name	21	400	8400
comes	16	500	8000
group	13	600	7800
lead	11	700	7700
friends	10	800	8000
begin	9	900	8100
family	8	1000	8000
brushed	4	2000	8000
sins	2	3000	6000

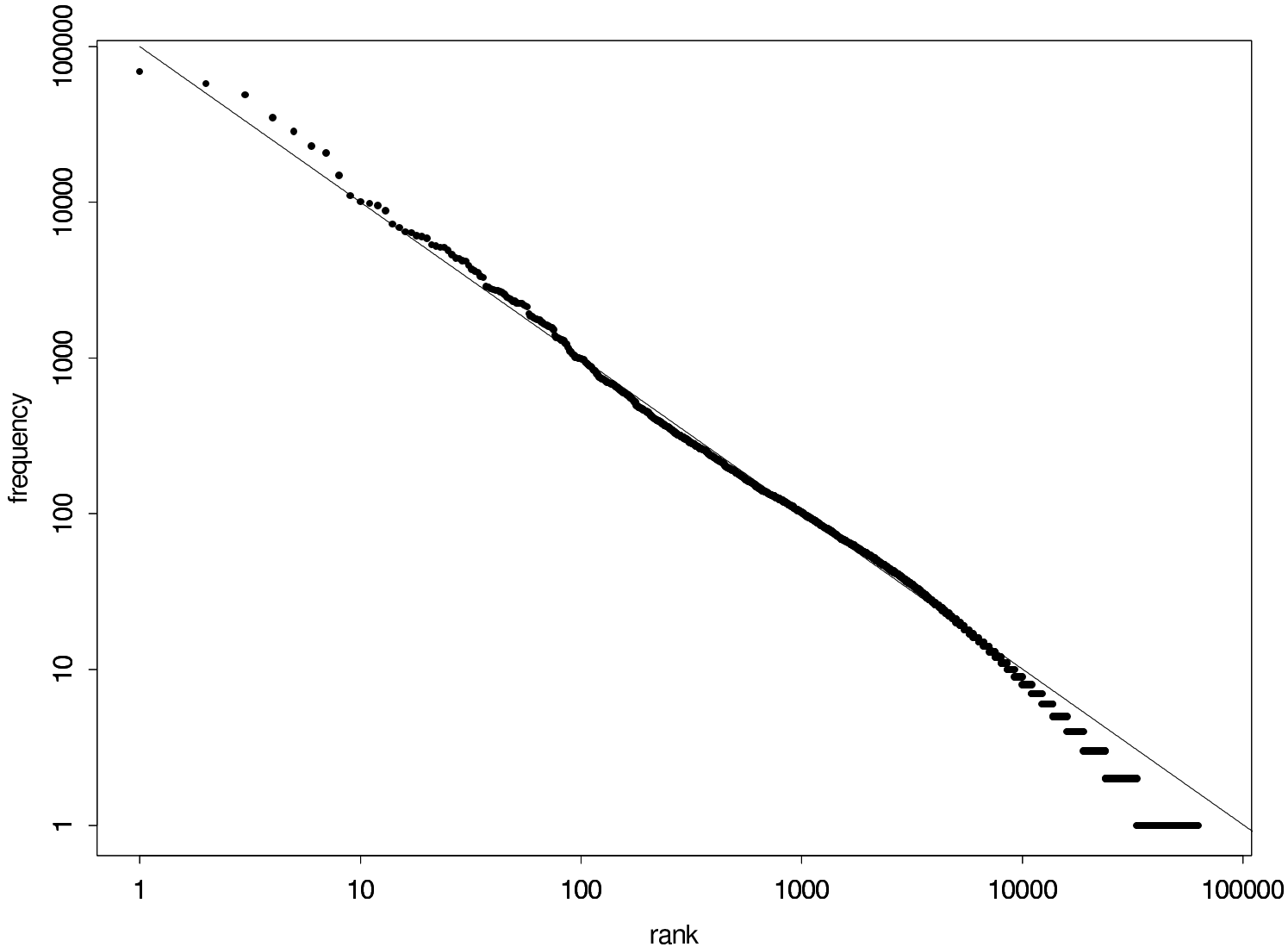
Zipf's law

Given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table (observation made by Harvard linguist George Kingsley Zipf).

Zipf's law states that: $f \propto \frac{1}{r}$

There is a constant k such that: $f \cdot r = k$.

Zipf's law for the Brown corpus



Zipf's law

According to Zipf's law:

- There is a very small number of very common words.
- There is a small-medium number of middle frequency words.
- There is a very large number of words that are infrequent.

(It's not fully understood why Zipf's law works so well for word frequencies.)
In fact, many other kinds of data conform closely to a Zipfian distribution:

- Populations of cities.
- Sizes of earthquakes.
- Amazon sales rankings.

Some tagging strategies

Now we have a sense that POS tagging is non-trivial. We'll look at several methods or strategies for automatic tagging.

- One simple strategy: just assign to each word its *most common tag*. (So **still** will *always* get tagged as an adverb — never as a noun, verb or adjective.) Call this *unigram* tagging, since we only consider one token at a time.
  Very, very important: compute the unigram frequency on different data from which you test the tagger. Why?
- Surprisingly, even this crude approach typically gives around 90% accuracy. (State-of-the-art is 96–98%).
- Can we do better? We'll look briefly at bigram tagging, then at Hidden Markov Model tagging.

Bigram tagging

Remember: linguistic tests often look at adjacent POS. (e.g. nouns are often preceded by determiners). Let's use this idea: we'll look at *pairs of adjacent POS*.
For each word (e.g. **still**), tabulate the frequencies of each possible POS *given the POS of the preceding word*.

Example (with made-up numbers):

still	DT	MD	JJ	...
NN	8	0	6	
JJ	23	0	14	
VB	1	12	2	
RB	6	45	3	

Given a new text, tag the words from left to right, assigning each word the most likely tag given the preceding one.

Could also consider trigram (or more generally n-gram) tagging, etc. But the frequency matrices would quickly get very large, and also (for realistic corpora) too 'sparse' to be really useful.

Problems with bigram tagging

- One incorrect tagging choice might have unintended effects:

	The	still	smoking	remains	of	the	campfire
Intended:	DT	RB	VBG	NNS	IN	DT	NN
Bigram:	DT	JJ	NN	VBZ	...		

- No lookahead: choosing the 'most probable' tag at one stage might lead to highly improbable choice later.

	The	still	was	smashed
Intended:	DT	NN	VBD	VBN
Bigram:	DT	JJ	VBD?	

We'd prefer to find the *overall most likely* tagging sequence given the bigram frequencies. This is what the Hidden Markov Model (HMM) approach achieves.

Hidden Markov Models

- The idea is to model the process by which words were generated using a probabilistic process.
- Think of the output as visible to us, but the internal states of the process (which contain POS information) as hidden.
- For some outputs, there might be several possible ways of generating them i.e. several sequences of internal states. Our aim is to compute the sequence of hidden states with the highest probability.
- Specifically, our processes will be 'FSTs with probabilities'. Simple, though not a very flattering model of human language users!

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh has a very rich history .

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh
NNP

$$p(NNP|\langle s \rangle) \times p(Edinburgh|NNP)$$

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh	has
NNP	VBZ

$$p(NNP|\langle s\rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh	has	a
NNP	VBZ	DT

$$p(NNP|\langle s\rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh	has	a	very
NNP	VBZ	DT	RB

$$p(NNP|\langle s\rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

$$p(RB|DT) \times p(very|RB)$$

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh	has	a	very	rich
NNP	VBZ	DT	RB	JJ

$$p(NNP|\langle s\rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

$$p(RB|DT) \times p(very|RB)$$

$$p(JJ|RB) \times p(rich|JJ)$$

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh	has	a	very	rich	history
NNP	VBZ	DT	RB	JJ	NN

$$p(NNP|\langle s\rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

$$p(RB|DT) \times p(very|RB)$$

$$p(JJ|RB) \times p(rich|JJ)$$

$$p(NN|JJ) \times p(history|NN)$$

Generating a Sequence

Hidden Markov models can be thought of as devices that generate sequences with hidden states:

Edinburgh	has	a	very	rich	history
NNP	VBZ	DT	RB	JJ	NN

$$p(NNP|\langle s\rangle) \times p(Edinburgh|NNP)$$

$$p(VBZ|NNP) \times p(has|VBZ)$$

$$p(DT|VBZ) \times p(a|DT)$$

$$p(RB|DT) \times p(very|RB)$$

$$p(JJ|RB) \times p(rich|JJ)$$

$$p(NN|JJ) \times p(history|NN)$$

$$p(STOP|NN)$$

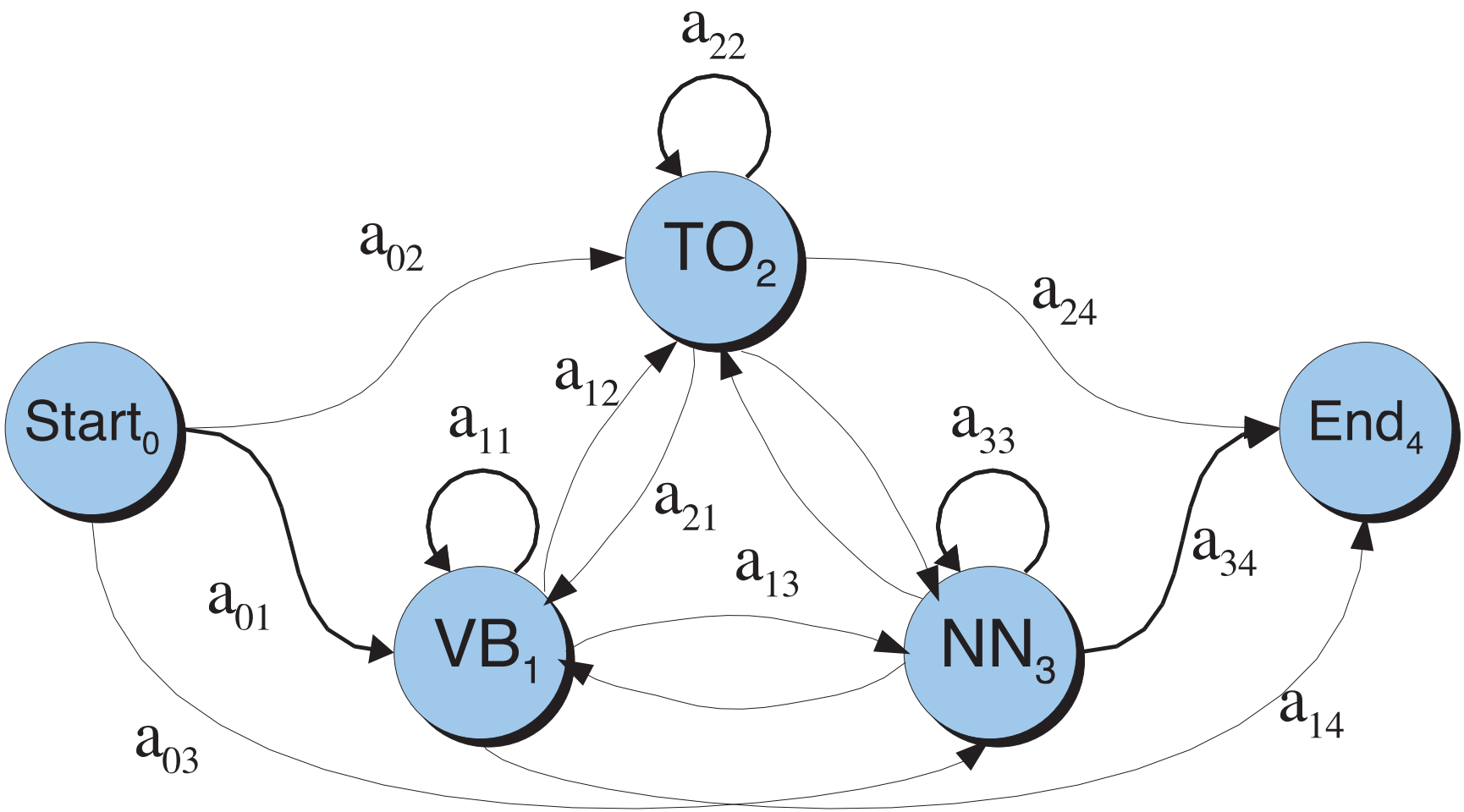
Definition of Hidden Markov Models

For our purposes, a Hidden Markov Model (HMM) consists of:

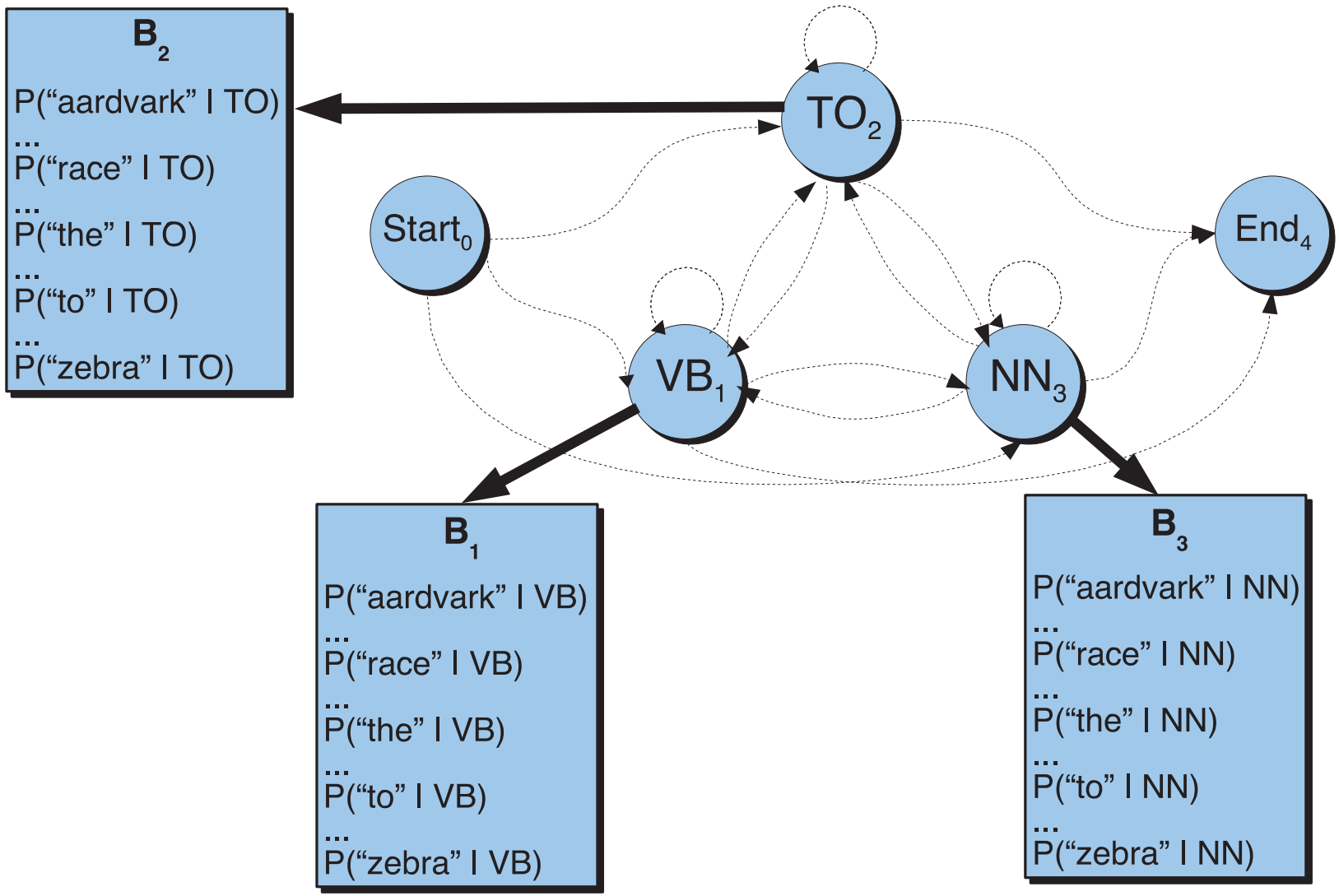
- A set $Q = \{q_0, q_1, \ldots, q_T\}$ of states, with q_0 the start state. (Our non-start states will correspond to *parts-of-speech*).
- A transition probability matrix $A = (a_{ij} \mid 0 \leq i \leq T, 1 \leq j \leq T)$, where a_{ij} is the probability of jumping from q_i to q_j. For each i, we require $\sum_{j=1}^{T} a_{ij} = 1$.
- For each non-start state q_i and word type w, an emission probability $b_i(w)$ of outputting w upon entry into q_i. (Ideally, for each i, we'd have $\sum_w b_i(w) = 1$.)

We also suppose we're given an observed sequence $w_1, w_2 \ldots, w_n$ of word tokens generated by the HMM.

Transition Probabilities



Emission Probabilities



Transition and Emission Probabilities

	VB	**TO**	**NN**	**PRP**
<s>	.019	.0043	.041	.67
VB	.0038	.035	.047	.0070
TO	.83	0	.00047	0
NN	.0040	.016	.087	.0045
PRP	.23	.00079	.001	.00014

	I	**want**	**to**	**race**
VB	0	.0093	0	.00012
TO	0	0	.99	0
NN	0	.000054	0	.00057
PRP	.37	0	0	0

How Do we Search for Best Tag Sequence?

We have defined an HMM, but how do we use it? We are given a **word sequence** and must find their corresponding **tag sequence**.

- It's easy to compute the probability of generating a word sequence $w_1 \ldots w_n$ via a specific tag sequence $t_1 \ldots t_n$: let t_0 denote the start state, and compute

$$\prod_{i=1}^{T} P(t_i|t_{i-1}).P(w_i|t_i) \tag{1}$$

  using the transition and emission probabilities.
- But how do we find the most likely tag sequence?

Question

Given n word tokens and a tagset with T choices per token, how many tag sequences do we have to evaluate?

1. $|T|$ tag sequences
2. n tag sequences
3. $|T| \times n$ tag sequences
4. $|T|^n$ tag sequences

Question

Given n word tokens and a tagset with T choices per token, how many tag sequences do we have to evaluate?

1. $|T|$ tag sequences
2. n tag sequences
3. $|T| \times n$ tag sequences
4. $|T|^n$ tag sequences

Bad news: there are $|T|^n$ sequences.

Question

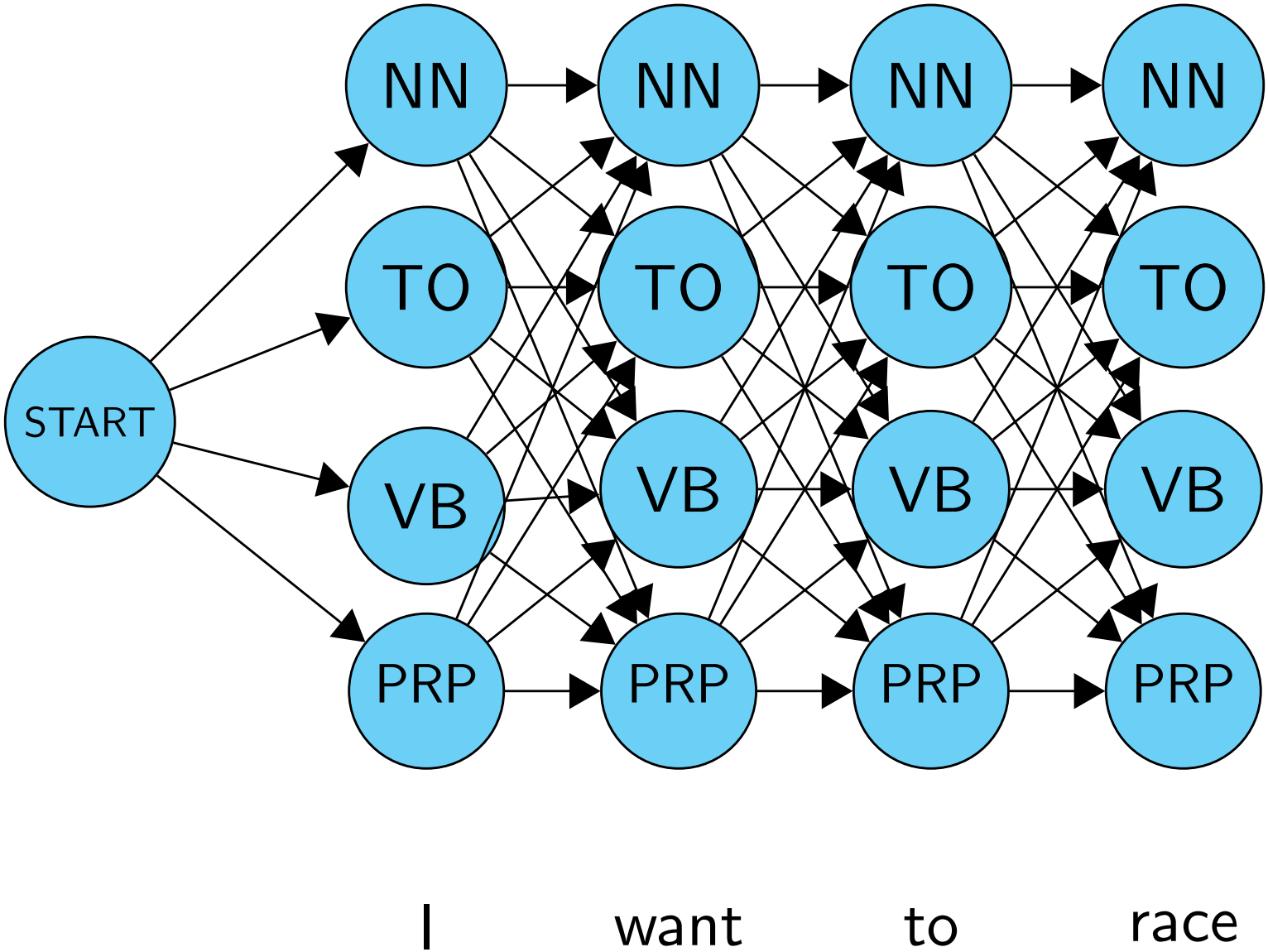
Given n word tokens and a tagset with T choices per token, how many tag sequences do we have to evaluate?

1. $|T|$ tag sequences
2. n tag sequences
3. $|T| \times n$ tag sequences
4. $|T|^n$ tag sequences

Bad news: there are $|T|^n$ sequences.

Good news: We can do this efficiently using **dynamic programming** and the **Viterbi algorithm**.

The HMM trellis



The Viterbi Algorithm

Keep a chart of the form $\mathrm{Table}(\mathrm{POS}, i)$ where POS ranges over the POS tags and i ranges over the indices in the sentence.

For all T and i:

$$\mathrm{Table}(T, i+1) \leftarrow \max_{T'} \mathrm{Table}(T', i) \times p(T|T') \times p(w_{i+1}|T)$$

and

$$\mathrm{Table}(T, 0) \leftarrow p(T|\langle s \rangle)$$

$\mathrm{Table}(., n)$ will contain the **probability** of the most likely sequence. To get the actual sequence, we need backpointers.

The Viterbi algorithm

Let's now tag the newspaper headline:

deal talks fail

Note that each token here could be a noun (N) or a verb (V).
We'll use a toy HMM given as follows:

	to N	to V
from start	.8	.2
from N	.4	.6
from V	.8	.2

Transitions

	deal	fail	talks
N	.2	.05	.2
V	.3	.3	.3

Emissions

The Viterbi matrix

	deal	talks	fail
N			
V			

	to N	to V
from start	.8	.2
from N	.4	.6
from V	.8	.2

Transitions

	deal	fail	talks
N	.2	.05	.2
V	.3	.3	.3

Emissions

$$\text{Table}(T, i+1) \leftarrow \max_{T'} \text{Table}(T', i) \times p(T|T') \times p(w_{i+1}|T)$$

The Viterbi matrix

	deal	talks	fail
N	.8x.2 = .16	$\leftarrow$ .16x.4x.2 = .0128 (since .16x.4 > .06x.8)	$\swarrow$ .0288x.8x.05 = .001152 (since .0128x.4 < 0.0288x.8)
V	.2x.3 = .06	$\nwarrow$ .16x.6x.3 = .0288 (since .16x.6 > .06x.2)	$\nwarrow$ .0128x.6x.3 = .002304 (since .0128x.6 > 0.0288x.2)

Looking at the highest probability entry in the final column and chasing the backpointers, we see that the tagging N N V wins.

The Viterbi Algorithm: second example

q_4	NN	0				
q_3	TO	0				
q_2	VB	0				
q_1	PRP	0				
q_o	start	**1.0**				
		<s>	I	want	to	race
			w_1	w_2	w_3	w_4

- For each state q_j at time i, compute
  $v_i(j) = \max_{k=1}^{n} v_{i-1}(k) a_{kj} b_j(w_i)$

The Viterbi Algorithm

q_4	NN	0				
q_3	TO	0				
q_2	VB	0				
q_1	PRP	0				
q_o	start	**1.0**				
		<s>	I	want	to	race
			w_1	w_2	w_3	w_4

1. Create probability matrix, with one column for each observation (i.e., word token), and one row for each non-start state (i.e., POS tag).
2. We proceed by filling cells, column by column.
3. The entry in column i, row j will be the **probability of the most probable route to state q_j that emits $w_1 \ldots w_i$.**

The Viterbi Algorithm

q_4	NN	0	$1.0 \times .041 \times 0$			
q_3	TO	0	$1.0 \times .0043 \times 0$			
q_2	VB	0	$1.0 \times .19 \times 0$			
q_1	PRP	0	$1.0 \times .67 \times .37$			
q_o	start	**1.0**				
		<s>	I	want	to	race
			w_1	w_2	w_3	w_4

- For each state q_j at time i, compute $v_i(j) = \max_{k=1}^{n} v_{i-1}(k) a_{kj} b_j(w_i)$
- $v_{i-1}(k)$ is **previous Viterbi path probability**, a_{kj} is **transition probability**, and $b_j(w_i)$ is **emission probability**.
- There's also an (implicit) **backpointer** from cell (i, j) to the relevant $(i-1, k)$, where k maximizes $v_{i-1}(k) a_{kj}$.

The Viterbi Algorithm

q_4	NN	0	0	$.025 \times .0012 \times 0.000054$		
q_3	TO	0	0	$.025 \times .00079 \times 0$		
q_2	VB	0	0	$.025 \times .23 \times .0093$		
q_1	PRP	0	**.025**	$.025 \times .00014 \times 0$		
q_0	start	**1.0**				
		<s>	I	want	to	race
			w_1	w_2	w_3	w_4

- For each state q_j at time i, compute $v_i(j) = \max_{k=1}^{n} v_{i-1}(k) a_{kj} b_j(w_i)$
- $v_{i-1}(k)$ is **previous Viterbi path probability**, a_{kj} is **transition probability**, and $b_j(w_i)$ is **emission probability**.
- There's also an (implicit) **backpointer** from cell (i, j) to the relevant $(i-1, k)$, where k maximizes $v_{i-1}(k) a_{kj}$.

The Viterbi Algorithm

q_4	NN	0	0	.000000002	.000053 × .047 × 0	
q_3	TO	0	0	0	.000053 × .035 × .99	
q_2	VB	0	0	**.00053**	.000053 × .0038 × 0	
q_1	PRP	0	**.025**	0	.000053 × .0070 × 0	
q_0	start	**1.0**				
		<s>	I	want	to	race
			w_1	w_2	w_3	w_4

- For each state q_j at time i, compute $v_i(j) = \max_{k=1}^{n} v_{i-1}(k) a_{kj} b_j(w_i)$
- $v_{i-1}(k)$ is **previous Viterbi path probability**, a_{kj} is **transition probability**, and $b_j(w_i)$ is **emission probability**.
- There's also an (implicit) **backpointer** from cell (i, j) to the relevant $(i-1, k)$, where k maximizes $v_{i-1}(k) a_{kj}$.

The Viterbi Algorithm

q_4	NN	0	0	.000000002	0	.0000018 × .00047 × .00057
q_3	TO	0	0	0	**.0000018**	.0000018×0×0
q_2	VB	0	0	**.00053**	0	.0000018×.83×.00012
q_1	PRP	0	**.025**	0	0	.0000018 × 0 × 0
q_0	start	**1.0**				
		<s>	I	want	to	race
			w_1	w_2	w_3	w_4

- For each state q_j at time i, compute $v_i(j) = \max_{k=1}^{n} v_{i-1}(k) a_{kj} b_j(w_i)$
- $v_{i-1}(k)$ is **previous Viterbi path probability**, a_{kj} is **transition probability**, and $b_j(w_i)$ is **emission probability**.
- There's also an (implicit) **backpointer** from cell (i, j) to the relevant $(i-1, k)$, where k maximizes $v_{i-1}(k) a_{kj}$.

The Viterbi Algorithm

q_4	NN	0	0	.000000002	0	4.8222e-13
q_3	TO	0	0	0	**.0000018**	0
q_2	VB	0	0	**.00053**	0	**1.7928e-10**
q_1	PRP	0	**.025**	0	0	0
q_0	start	**1.0**				
		<s>	I	want	to	race
			w_1	w_2	w_3	w_4

- For each state q_j at time i, compute $v_i(j) = \max_{k=1}^{n} v_{i-1}(k) a_{kj} b_j(w_i)$
- $v_{i-1}(k)$ is **previous Viterbi path probability**, a_{kj} is **transition probability**, and $b_j(w_i)$ is **emission probability**.
- There's also an (implicit) **backpointer** from cell (i, j) to the relevant $(i-1, k)$, where k maximizes $v_{i-1}(k) a_{kj}$.

Connection between HMMs and finite state machines

Hidden Markov models are finite state machines with probabilities added to them.

If we think of finite state automaton as generating a string when randomly going through states (instead of scanning a string), then hidden Markov models are such FSMs where there is a specific probability for generating each symbol at each state, and a specific probability for transitioning from one state to another.

As such, the Viterbi algorithm can be used to find the most likely sequence of *states* in a probabilistic FSM, given a specific input string.

Question: where do the probabilities come from?

Example Demo

`http://nlp.stanford.edu:8080/parser/`

- Relies both on "distributional" and "morphological" criteria
- Uses a model similar to hidden Markov models