Handout 11: Part-of-speech Tagging

Tagged Text

1. Using a tagger out of the box

```
>>> from nltk import word_tokenize, pos_tag
>>> words = word_tokenize('The cat chased him')
>>> pos_tag(words)
[('The', 'DT'), ('cat', 'NN'), ('chased', 'VBD'), ('him', 'PRP')]
```

2. Manually tagged text

```
>>> from nltk.corpus import brown
>>> brown.tagged_words()[:7]
[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand',
'JJ-TL'), ('Jury', 'NN-TL'), ('said', 'VBD'), ('Friday', 'NR')]
```

3. Simplified tagset

a. "Universal tagset"

```
>>> brown.tagged_words(tagset='universal')
[('The', 'DET'), ('Fulton', 'NOUN'), ('County', 'NOUN')
('Grand', 'ADJ'), ('Jury', 'NOUN'), ('said', 'VERB'), ...]
```

b. Complete list:

ADJ	*recent*	CONJ	*and, that*	NUM	*two*	PUNC	.
ADP	*in*	DET	*the*	PRON	*it*	VERB	*broke, should*
ADV	*often*	NOUN	*jury*	PRT	*up*	X	*avec*

4. Exploring tags

a. Get tag pairs

```
>>> from nltk import bigrams, FreqDist
>>> twords = brown.tagged_words(categories='learned')
>>> tpairs = list(bigrams(t for (w,t) in twords))
```

b. Which is most frequent?

```
>>> fd = FreqDist(tpairs)
>>> fd.most_common(3)
[(('NN', 'IN'), 8451), (('AT', 'NN'), 8323), (('IN', 'AT'), 7781)]
```

c. Which occur after AT?

```
>>> cfd = ConditionalFreqDist(tpairs)
>>> cfd['AT'].most_common(3)
[('NN', 8323), ('JJ', 3715), ('NNS', 1276)]
```

d. What are some examples of “IN NN” pairs?

```
>>> exs = [(w1,w2) for ((w1,t1),(w2,t2)) in bigrams(twords)
...    if t1 == 'IN' and t2 == 'NN']
>>> exs[:3]
[('of', 'information'), ('at', 'wave'), ('by', 'coincidence')]
```

5. Building taggers

a. Default tagger: always assign “NN”

```
>>> t0 = DefaultTagger('NN')
>>> t0.tag(words)
[('The', 'NN'), ('cat', 'NN'), ('chased', 'NN'), ('him', 'NN')]
```

b. Regex tagger: look at suffixes

```
>>> ptns = [(r'.*ing$', 'VBG'), (r'.*ed$', 'VBD')]
>>> t1 = RegexpTagger(ptns)
>>> t1.tag(words)
[('The', None), ('cat', None), ('chased', 'VBD'), ('him', None)]
```

c. Backing off

```
>>> t1 = RegexpTagger(ptns, backoff=t0)
>>> t1.tag(words)
[('The', 'NN'), ('cat', 'NN'), ('chased', 'VBD'), ('him', 'NN')]
```

d. Unigram tagger

```
>>> model = {'the':'AT', 'him':'PPO'}
>>> t2 = UnigramTagger(model=model, backoff=t1)
>>> t2.tag(words)
[('The', 'NN'), ('cat', 'NN'), ('chased', 'VBD'), ('him', 'PPO')]
```

e. Training: set aside some for testing

```
>>> sents = brown.tagged_sents(categories='news')
>>> n = int(.9 * len(sents))
>>> train = sents[:n]
>>> test = sents[n:]
```

f. Train a unigram tagger

```
>>> t2 = UnigramTagger(train, backoff=t1)
>>> t2.tag(words)
[('The', 'AT'), ('cat', 'NN'), ('chased', 'VBD'), ('him', 'PPO')]
```

g. Evaluation

```
>>> t0.evaluate(test)
0.1262832652247583
>>> t1.evaluate(test)
0.15070268115219776
>>> t2.evaluate(test)
0.8436160669789694
```

h. Bigram, trigram taggers

```
>>> t3 = BigramTagger(train, backoff=t2)
>>> t3.evaluate(test)
0.85418120203329018
>>> t4 = TrigramTagger(train, backoff=t3)
>>> t4.evaluate(test)
0.852387122495764
```

i. Why did the performance go down?

Bigram HMM Taggers

6. How a bigram HMM tagger works

a. Finite-state automaton with outputs from states

b. Transitions:

	#	NNS	RB	VB
#	0	.75	.13	.13
NNS	.25	.17	.17	.42
RB	.50	.25	.0	.25
VB	.43	.43	.14	.0

	#	NNS	RB	VB
#	∞	.1	.9	.9
NNS	.6	.8	.8	.4
RB	.3	.6	∞	.6
VB	.4	.4	.8	∞

c. Emissions:

	bark	cats	dogs	often
NNS	.17	.42	.42	0
RB	0	0	0	1.0
VB	.71	0	.29	0

	bark	cats	dogs	often
NNS	.8	.4	.4	∞
RB	∞	∞	∞	0
VB	.1	∞	.5	∞

d. Cost of state-sequence + output

.1	NNS	.8	NNS	.8	RB	.3	3.2
	.4		.8		0		
.1	NNS	.4	VB	.8	RB	.3	**2.1**
	.4		.1		0		
.9	VB	.4	NNS	.8	RB	.3	3.0
	.5		.8		0		
.9	VB	∞	VB	.8	RB	.3	∞
	.5		.1		0		
	dogs		bark		often		

7. Training

a. For each row in transition/emission tables, count how many times each transition/emission occurs in training

	#	NNS	RB	VB	
NNS	3	2	2	5	12

b. Normalize to make it a probability distribution

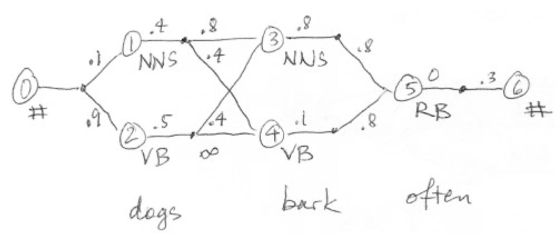
	#	NNS	RB	VB
NNS	3/12	2/12	2/12	5/12

c. That is:

	#	NNS	RB	VB
NNS	.25	.17	.17	.42

8. Viterbi algorithm: finding best sequence efficiently

a. Configuration graph



b. We construct the best path we can, working left to right

c. Consider partial path x up to node N, and partial path y from N to the end:

$$c(xy) = c(x) + c(y)$$

d. Key idea: consider partial paths x_1, x_2 leading up to node N. No matter what completion y we choose (from n to the end), we have:

$$c(x_1) + c(y) < c(x_2) + c(y) \qquad \text{if } c(x_1) < c(x_2)$$

e. For each node, what is the best incoming path, and what is its score?

N		**Cands**	**BestPrev**	**Cost**
1			0	.1
2			0	.9
3	1	$(.1) + .4 + .8 = 1.3$	1	1.3
	2	$(.9) + .5 + .4 = 1.8$		
4	1	$(.1) + .4 + .4 = .9$	1	.9
	2	$(.9) + .5 + \infty = \infty$		
5	3	$(1.3) + .8 + .8 = 2.9$	4	1.8
	4	$(.9) + .1 + .8 = 1.8$		
6	5	$(1.8) + 0 + .3 = 2.1$	5	2.1

9. The heart of the algorithm: scoring a node n

a. Get a list of candidate pairs (s, p) where p is a preceding node and s is the **score through** p to n

b. Pick the best, record the **best preceding** node and the score for n

10. The score through p to n consists of: the score of p, the cost of the emission at p, and the cost of the transition from p to n.

11. The overall algorithm

- **a.** Input: a list of word tokens
- **b.** Tagger also has a model available (transition and emission tables)
- **c.** Build the graph: create one node for each POS of each word token.
- **d.** A `Node` has a word token, a POS, a list of preceding nodes. We will compute a score and a best-previous node.
- **e.** Pass through the list of nodes. Compute the score and best-previous for each.
- **f.** Unwind: read off the tag sequence.

12. Unwinding:

- **a.** Follow the trail of best-previous nodes backwards from end node. Read off the POS of each node.
- **b.** Our example:

$$\begin{array}{c} 6 \\ \# \end{array} \rightarrow \begin{array}{c} 5 \\ \text{RB} \end{array} \rightarrow \begin{array}{c} 4 \\ \text{VB} \end{array} \rightarrow \begin{array}{c} 1 \\ \text{NNS} \end{array} \rightarrow \begin{array}{c} 0 \\ \# \end{array}$$

Turning it around right-ways: (dogs) NNS (bark) VB (often) RB