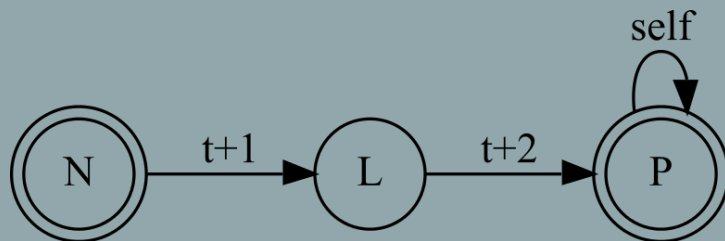


LING-362

Introduction to Natural Language Processing

Tagging and Hidden Markov Models II



The PTB tag set (vanilla)

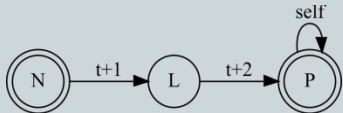
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

PRP\$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VCN	Verb, past participle
VBP	Verb, non-3rd person sg. present
VBZ	Verb, 3rd person sg. present
WDT	Wh-determiner
WP	Wh-pronoun
WP\$	Possessive wh-pronoun
WRB	Wh-adverb

Tagging exercise

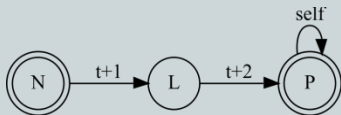
◎ *Is ISIS Going Broke?*

- Solution in Canvas



The PTB tag set

- ⊙ There is a lot to be said about the PTB tag set
 - Successes and shortcomings
 - Extensions since its inception – notably through the AMALGAM project (2001), TreeTagger, OntoNotes, English Web Treebank...
- ⊙ We don't have time to discuss these...
- ⊙ For this course: PTB (a.k.a. vanilla PTB) will be our only tag set for English (more: LING-367)



How does a POS tagger work?

- ◉ To decide what POS tags to assign, an automatic tagger consults training data
 - Known POS distributions
 - Known conditional probabilities $P(POS2/POS1)$
 - ...
- ◉ We can get initial probabilities for a single word...
 - but the tagger can also continue a plausible sequence
 - First let's see what we can do with a tagger using NLTK



Tagging with NLTK – tagging.py

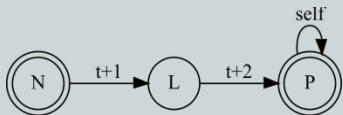
```
import nltk  
text = "Mr. Pickwick turned azure."
```

```
# Get a list of tokens  
tokenized = nltk.word_tokenize(text)
```

```
# Make it a list of (token, pos) tuples  
tagged = nltk.pos_tag(tokenized)
```

```
print(tagged)  
# Note the error!
```

```
[('Mr.', 'NNP'), ('Pickwick', 'NNP'), ('turned', 'VBD'),  
 ('azure', 'NN'), ('.', '.')] ]
```



What can we do with 'tagged'?

- ◉ Tagged data is a **list** of **tuples**
- ◉ Can be separated into **two lists** for processing just tags:

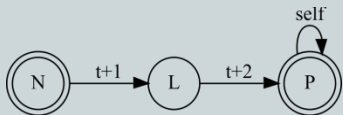
```
words = []
```

```
tags = []
```

```
for word, tag in tagged:
```

```
    words.append(word)
```

```
    tags.append(tag)
```



Inspecting frequencies

```
tag_dist = nltk.FreqDist(tags)
word_dist = nltk.FreqDist(words)
print(tag_dist.most_common(4))
print(word_dist.most_common(100)[90:])
```



And plotting

- To plot you must install Matplotlib and NumPy from the command line:

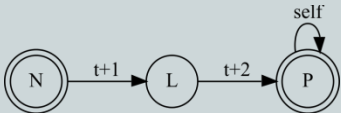
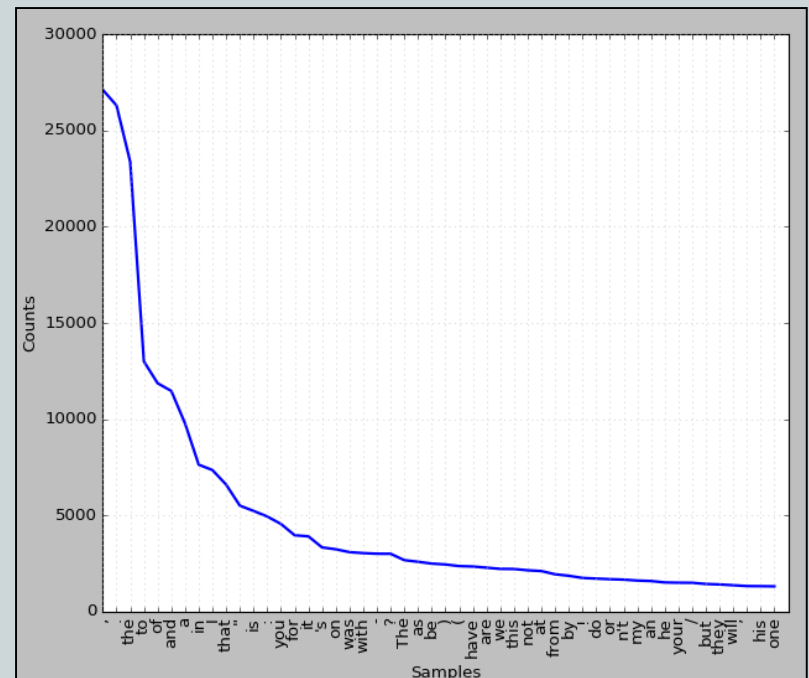
> *pip install numpy*

> *pip install matplotlib*

- Then in Python:

```
word_dist.plot(50)
```

```
word_dist.plot(50, cumulative=True)
```



Tagged data from nltk

- ◎ You can also get some ready corpora from NLTK:

```
from nltk.corpus import masc_tagged
```

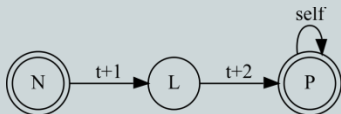
```
words = []
```

```
tags = []
```

```
for word, tag in masc_tagged.tagged_words():
```

```
    words.append(word)
```

```
    tags.append(tag)
```



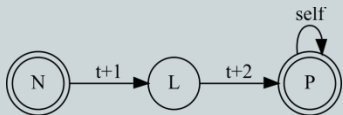
Homework – for Monday, Nov 8

◎ Write a program that:

- Takes a text file input
- Tokenizes and tags the text with NLTK
- Goes through the result and saves **only common nouns**
- Prints the most frequent 10
- Plots the top 100 using nltk's FreqDist

◎ Run your program on some text

◎ Find one word mistagged as a noun and write as a comment – why do you think this happened?



Exercise – verb proportions

◎ Try to get the proportion of verbs in some text:

- Read a text file saved from the Web
- Tokenize and tag it
- Loop through data and get:
 - Length in tokens
 - Amount of verbs
 - Print proportions with a verbal tag using one of:

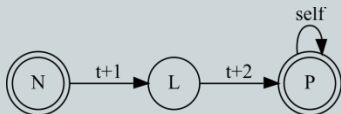
```
if tag in ["VB", "VBZ", ...]:
```

```
...
```

```
if re.match(r'V.*', tag) is not None:
```

```
...
```

```
if tag.startswith("V"):
```



We can also get this by sentence

```
import argparse  
from nltk import word_tokenize, pos_tag,  
sent_tokenize
```

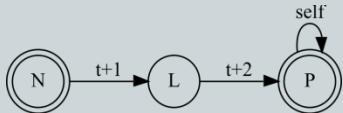


We can also get this by sentence

```
parser = argparse.ArgumentParser()
parser.add_argument("file")
options = parser.parse_args()
```

```
with open(options.file, 'r') as f:
    text = f.read()
```

```
sentences = sent_tokenize(text)
sent_num = 0
```



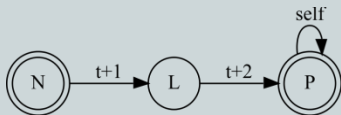
We can also get this by sentence

```
for sentence in sentences:
    tokens = nltk.word_tokenize(sentence)
    tagged = nltk.pos_tag(tokens)
    length = len(tokens)
    verbs = 0
    sent_num += 1
    for token, tag in tagged:
        if tag.startswith('V'):
            verbs += 1
    print("Verb ratio for S" + str(sent_num) + ": " +
          str(float(verbs)/length))
```



How does tagging work?

- ◉ Why did NLTK mis-tag *azure*?
- ◉ It seems the following is probably true in NLTK's training data:
 - Known POS distributions: $P(NN) > P(JJ)$
 - Conditional probabilities:
 $P(NN/NNP,VBD) > P(JJ/NNP,VBD)$
 - ...
- ◉ But how does NLTK know that *azure* was preceded by VBD+NNP???



HMMs

⊙ An HMM is really a **weighted FSA**

⊙ The HMM definition comprises:

- $V = v_1 \dots v_V$
- $Q = q_1, \dots q_N \quad (q_0, q_F)$
- $A = a_{11}, a_{12} \dots a_{n1} \dots a_{nn}$
- $O = \langle o_1, \dots, o_T \rangle$
- $B = b_i(o_t)$

input vocabulary items

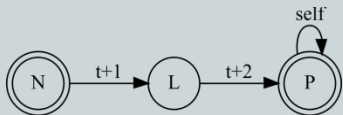
states

transition prob. matrix

ordered observations of V

prob. of o_t given q_i

a.k.a 'emission' probs.

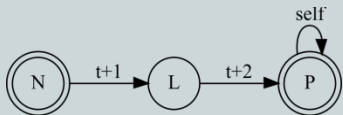


HMMs – formal definition

- Transition probabilities (A):
(adapted from Jurafsky & Martin)

	Q0	VB	TO	NN	QF
Q0	--	0.0004	0.0064	0.0365	0
VB	--	0.0038	0.035	0.047	0.012
TO	--	0.83	0	0.00047	0.00079
NN	--	0.0040	0.016	0.087	0.23
QF	--	--	--	--	--

$P(\text{VB} | \text{TO}) = 0.83$ (rows give the condition)

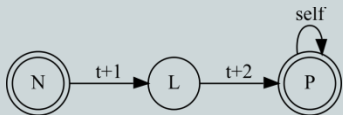


HMMs – formal definition

- Emission probabilities (B):
(adapted from Jurafsky & Martin 2008)

	I	want	to	see
VB	0	0.0093	0	0.12
TO	0	0	0.99	0
NN	0	0.000054	0	0.000007

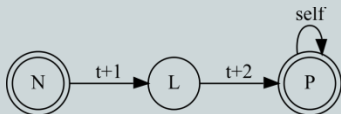
$P(\text{see} | \text{VB}) = 0.12$ (assuming this is VB, chance to get 'see')



Where are we in the definition?

◎ The POS tagging task maps directly to the HMM definition:

- V: words of the English language
- Q: the parts of speech (state: DT \rightarrow state: NN)
- A: probability of DT \rightarrow NN, ... (Table A)
- B: probability $P(\text{the} \mid \text{DT})$, ... (Table B)
- O: The observed text to be tagged $\langle w_1, \dots, w_n \rangle$



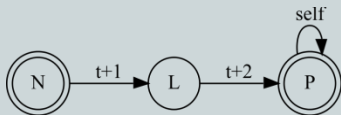
Using the chain

- ⊙ Given A and B, it's not complicated to get the single next most probable tag
- ⊙ **But...**
 - What if choosing that tag will lead us to very unlikely choices later on?
 - What if choosing the second best one now is better in total?
 - Need to traverse multiple paths and remember probabilities – hard to do efficiently



The Viterbi algorithm

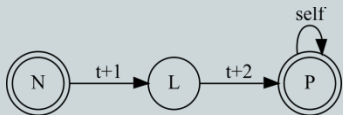
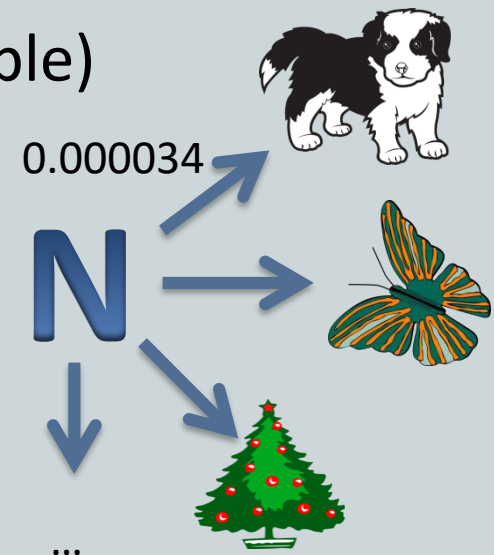
- ⊙ Devised multiple times in parallel, including by Andrew Viterbi
- ⊙ Essential for POS tagging but also:
 - Signal processing (cellphone signal decoding)
 - DNA sequencing
 - WiFi error correction
 - ... and much more
- ⊙ A special case of **dynamic programming** (contrast: **greedy algorithm**)



The Viterbi algorithm

◉ What we'll need for the algorithm:

- Table A: a **dictionary** of transition probabilities
(somedict[DT][NN] -> probability of transition DT|NN)
- Table B: a **dictionary** of word|tag probabilities
(otherdict[N][puppy] = 0.000034)
- State space (i.e. the tag set, list or tuple)
- Start probability dictionary for q0
(start[DT] = 0.1812)



Building table B – emit_p

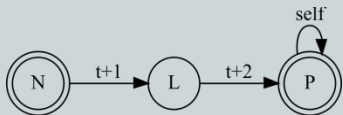
- ◉ We can write code to make sure we have a value for every possible item in a dictionary
- ◉ But what about OOV items?
- ◉ Example: $p(\text{dancerliness}) = ??$
 - `emit_p['NN']['dancerliness']`
`KeyError: 'dancerliness'`
- ◉ Solvable using some **if ... :**



Building block: defaultdict

- ⦿ A better way is to have a dictionary that knows how to initialize unseen values
- ⦿ Uses a **default** value if key is unknown:
 - Should be initializable with a data type
 - Or a function returning some value

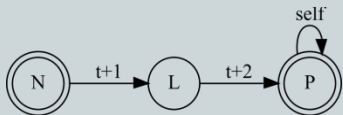
```
from collections import defaultdict  
my_dict = defaultdict(int)  
print(my_dict["puppy"])  
0
```



Building block: defaultdict

- ⦿ How to return a specific value?
- ⦿ Suppose we want the default to be 0.5
- ⦿ Instead of int, we can give a special function as the default:

Suppose half_returner is a function, always returns 0.5
`my_dict = defaultdict(half_returner)`



Building block: defaultdict

- ◎ This is a little cumbersome, so a more Pythonic way is this to use the anonymous function **lambda**:

```
my_dict = defaultdict(lambda: 0.5)
```

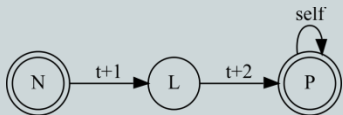
- ◎ Same as:

Suppose half_returner is a function, always returns 0.5

```
def half_returner():
```

```
    return 0.5
```

```
my_dict = defaultdict(half_returner)
```



Building block: defaultdict

⊙ And we can even make a defaultdict of defaultdicts:

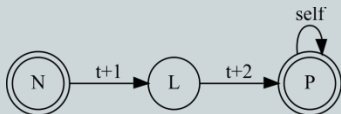
- `x = defaultdict(lambda: defaultdict(lambda: 0.00000001)))`

⊙ Now x is a dictionary:

- which defaults unknown entries to dictionaries
 - which default unknown entries to 0.00000001..

➤ So what is the value of:

`x["NN"]["puppy"]` ?



Viterbi - implementation

- Download and debug

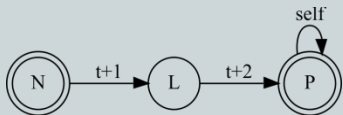
- tagging/viterbi_simple.py*

- Imports and states:

```
from collections import defaultdict
```

```
# Tagger states Q (the tag set)
```

```
states = (',', 'CC', 'CD', 'DT', 'EX', 'IN', 'JJ', 'JJR', 'JJS', 'LS', 'MD', 'NN', 'NNP', 'NNPS',  
'NNS', 'PDT', 'PRP', 'PRP$', 'RB', 'RBR', 'RBS', 'SENT', 'SYM', 'TO', 'UH', 'VB', 'VBD',  
'VBG', 'VBP', 'VBZ', 'WDT', 'WP', 'WRB')
```



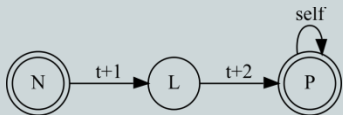
Viterbi – starting probabilities

Training data:

q0 probabilities:

can't use prior tags to estimate initial state

```
start_p = {  
    ',': 0.006,  
    'CC': 0.0451,  
    'CD': 0.0158,  
    'DT': 0.1365,  
    'EX': 0.0102,  
    ...  
}
```



Viterbi – transitional probabilities

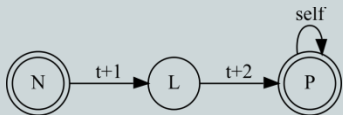
- Training data for transition is a dictionary:

`trans_p = {}`

- We could set the transition probabilities by assigning nested dictionaries:

`trans_p['DT'] = {'JJ': 0.0208, 'NN': 0.0397 ...}`

- But these would be regular dictionaries, no default value for missing values



Viterbi - transitional probabilities

◎ A better way:

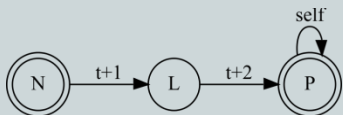
```
trans_p =  
defaultdict(lambda: defaultdict(lambda: 0.00000001))
```

Don't make a new dictionary,

just update the defaultdict with new dictionary values

```
trans_p['DT'].update({'JJ': 0.0208, 'NN': 0.0397, ...})
```

```
trans_p['IN'].update({',': 0.0015, 'CD': 0.0016, ...})
```



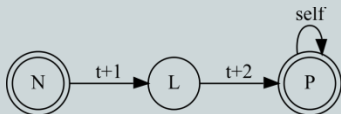
Viterbi – emission probabilities

```
emit_p = defaultdict(lambda: defaultdict(lambda:  
0.00000001))
```

```
emit_p['VB']['want'] = 0.0093
```

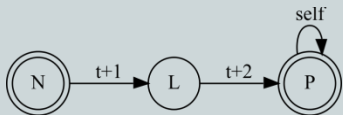
```
emit_p['VB']['fly'] = 0.0001
```

...



The algorithm 1/3

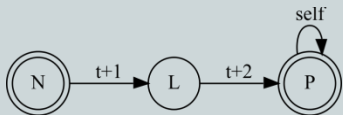
```
def viterbi(obs, states, start_p, trans_p, emit_p):  
    path = [{}] # The Viterbi path is a list of dicts mapping tok+tag to probability  
  
    # Get initial probabilities for each tag given first token (obs[0])  
    for tag in states:  
        path[0][tag] = start_p[tag]*emit_p[tag][obs[0]]
```



The algorithm 2/3

Get subsequent probabilities for obs[t] where $t > 0$ (tokens after the first)

```
for tok_num in range(1, len(obs)):
    path.append({})
    backpointer = {}
    for tag in states:
        max_prob = 0.0
        probs = []
        for prev_tag in states:
            probs.append(path[tok_num - 1][prev_tag] * trans_p[prev_tag][tag] * emit_p[tag][obs[tok_num]])
            if prob > max_prob:
                max_prob = prob
                best_prev = prev_tag
        path[tok_num][tag] = max_prob
        backpointer[tag] = best_prev
    backpath.append(backpointer)
```



The algorithm 3/3

```
optimal_list = []
```

```
# Go through each token position in path;
```

```
# Each token position is now a dictionary
```

```
# of tags to continuation probabilities given previous context
```

```
current_best_tag = max(path[-1], key=path[-1].get)
```

```
optimal_list.append(current_best_tag)
```

```
backpath.reverse()
```

```
for backpointer in backpath:
```

```
    optimal_list.append(backpointer[current_best_tag])
```

```
    current_best_tag = backpointer[current_best_tag]
```

```
optimal_list.reverse()
```

```
# The highest probability
```

```
max_total_prob = max(path[-1].values())
```

```
print('Best sequence: ' + ' '.join(optimal_list) + ' with highest probability of ' +  
str(float(max_total_prob)))
```

