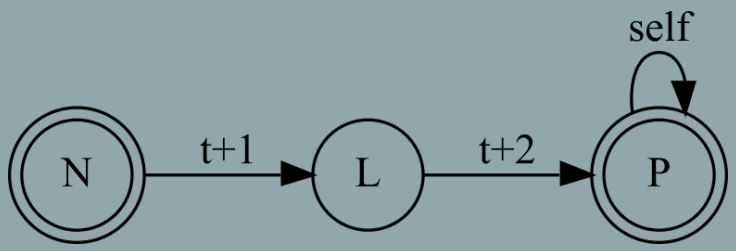
LING-362

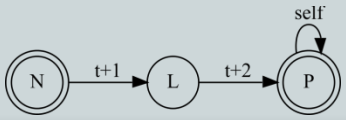
Introduction to **N**atural **L**anguage **P**rocessing

Viterbi Decoding



HMMs

- An HMM is really a **weighted FSA**
- The HMM definition comprises:
  - $V = v_1 \ldots v_V$ # input vocabulary items
  - $Q = q_1, \ldots q_N \quad (q_0, q_F)$ # states
  - $A = a_{11}, a_{12} \ldots a_{n1} \ldots a_{nn}$ # transition prob. matrix
  - $O = \langle o_1, \ldots, o_T \rangle$ # ordered observations of V
  - $B = b_i(o_t)$ # prob. of o_t given q_i
    # a.k.a 'emission' probs.

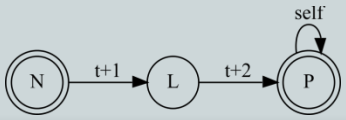


HMMs – formal definition

- 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(TO|VB) = 0.035 (rows give the condition)

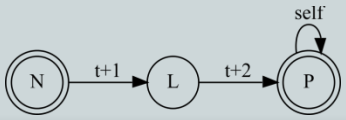


HMMs – formal definition

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

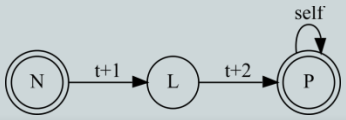
	I	want	to	see
VB	0	0.0093	0	0.12
TO	0	0	0.99	0
NN	0	0.000054	0	0.000007

P(see|VB) = 0.12 (assuming this is VB, chance to get 'see')



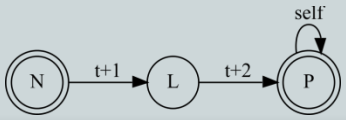
Where are we in the definition?

- The POS tagging task maps directly to the HMM definition:
  - V: words of the English language
  - Q: the parts of speech (state: DT -> state: NN)
  - A: probability of DT -> NN, … (Table A)
  - B: probability P(the|DT), … (Table B)
  - O: The observed text to be tagged <w_1, …, w_n>



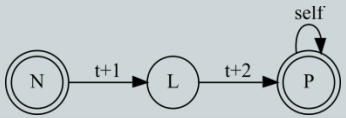
Using the chain

- 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



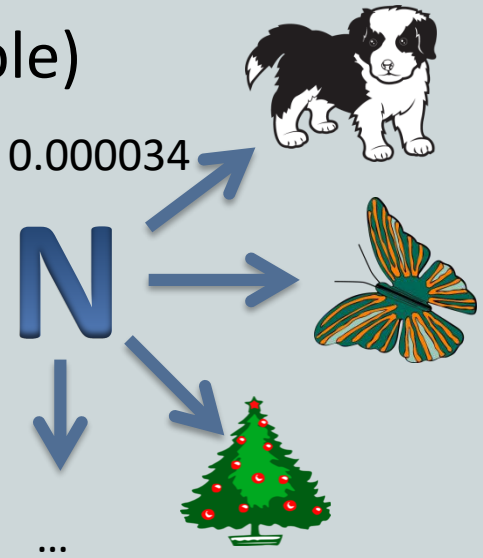
The Viterbi algorithm

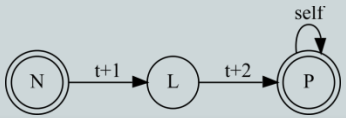
- Devised multiple times in parallel, including by Andrew Viterbi
  - Essential for POS tagging but also:
  - Signal processing (cell phone signal decoding)
  - DNA sequencing
  - WiFi error correction
  - … and much more
- A special case of **dynamic programming** (contrast: **greedy algorithm**)



The Viterbi algorithm

- What we'll need for the algorithm:
  - Table A: a **dictionary** of transition probablities (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)



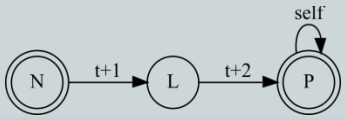


Building table B – emit_p

- 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(dancerliness) = ??
  - emit_p['**NN**']['**dancerliness**']

  KeyError: 'dancerliness‘

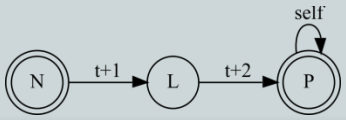
- Solvable using some **if … :**



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

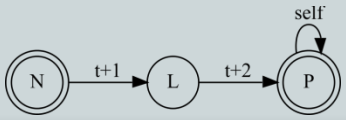
```
from collections import defaultdict
my_dict = defaultdict(int)
print(my_dict["puppy"])
0
```



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)
```



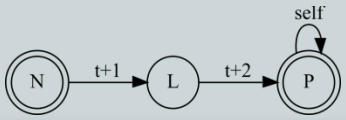
Building block: defaultdict

- 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:

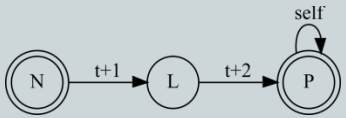
```
# Suppose half_returner is a function, always returns 0.5
def half_returner():
    return 0.5
my_dict = defaultdict(half_returner)
```



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 0.0000001..
- So what is the value of:

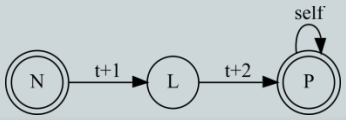
  x[**"puppy"**][**"the"**] ?



Viterbi - implementation

- Download and debug *tagging/viterbi_simple.py*
- Imports and states:

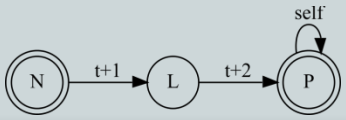
```
from collections import defaultdict


# 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

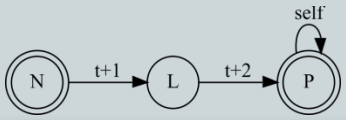
- Training data for transitions is a dictionary:

  trans_p ={}

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

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

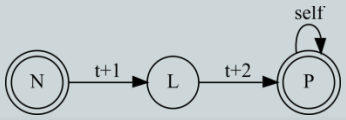
- But these would be regular dictionaries, no default value for missing keys



Viterbi - transitional probabilities

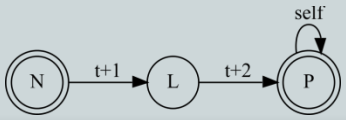
- A better way:

```
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, …}
```



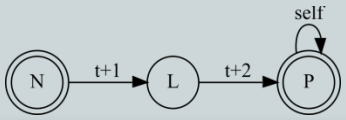
Viterbi – emission probabilities

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

emit_p['VB']['want'] = 0.0093
emit_p['VB']['fly'] = 0.0001
...
```



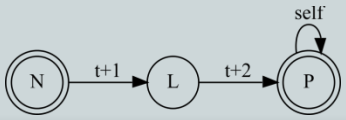
The algorithm 1/3

```
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]]
```



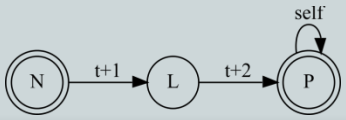
The algorithm 2/3

```
# 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)
```

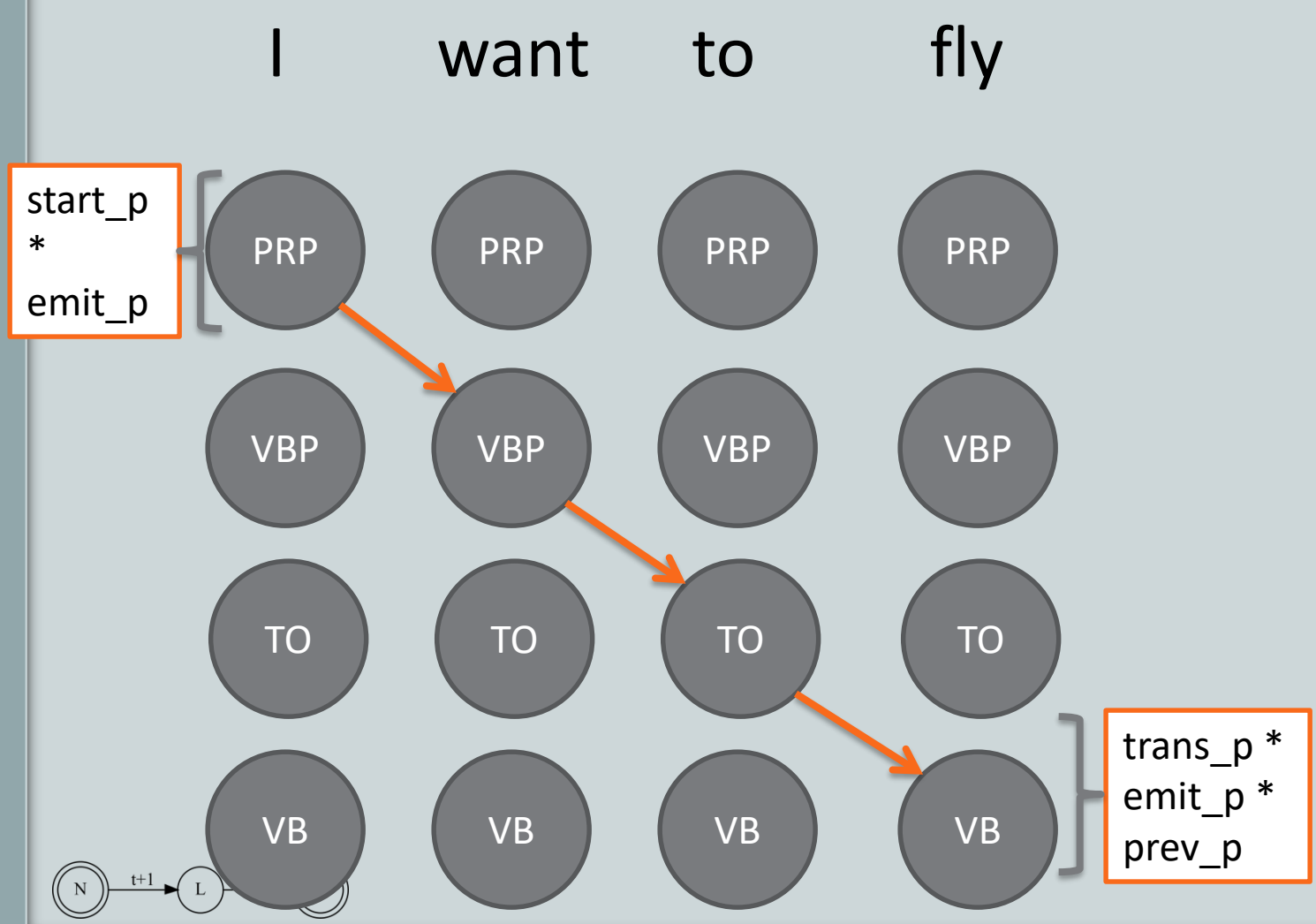


The algorithm 3/3

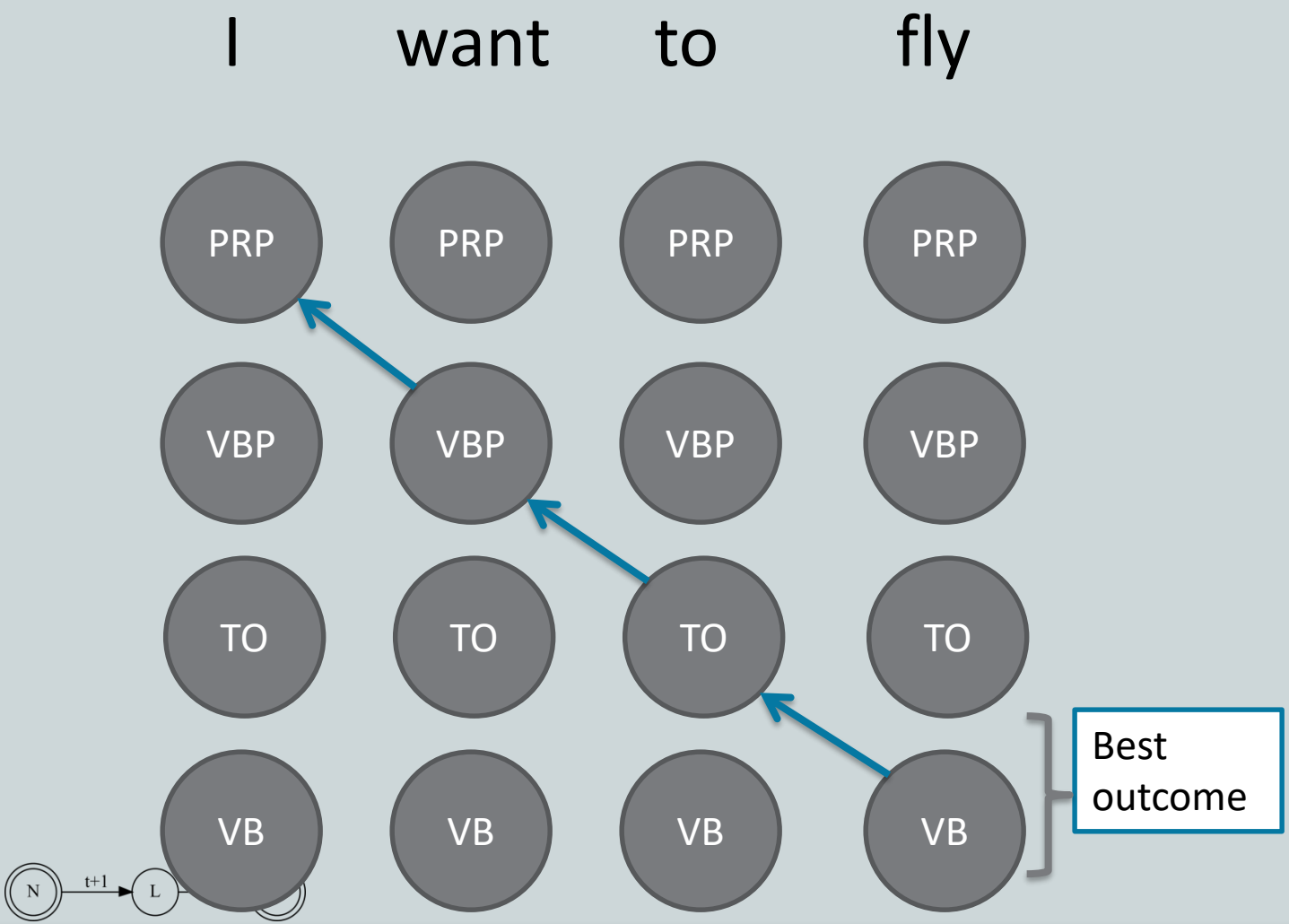
```
optimal_list = []

# 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)))
```



Viterbi algorithm



Backtrace step



Group work

- Let's try to trip up and then fix our tagger:
  - Split up into groups
  - Pick a sentence – not too long, about 4-7 words
    - Does the tagger work right?
    - How could you fix it?
    - Let each member try a minor variation on this sentence – can the fixes work without breaking other variations?
  - Add emission probabilities for new words
    - Put them here: https://corpling.uis.georgetown.edu/etherpad/p/viterbi
    - You can make them up or use a corpus: https://corpling.uis.georgetown.edu/cqp/
    - The TAs and I will provide guidance