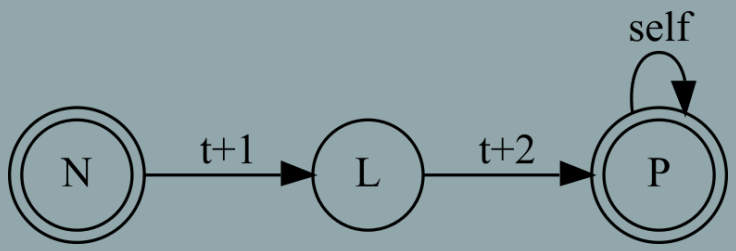
LING-362

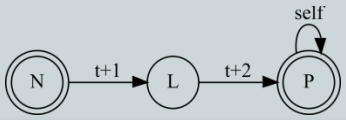
Introduction to Natural Language Processing

Viterbi (ctd.) and Sequence Labeling



HMMs

- An HMM is really a **weighted FSA**
- The HMM definition comprises:
  - $V = v_1 \ldots v_V$ # for us: English words
  - $Q = q_1, \ldots q_N \quad (q_0, q_F)$ # for us: possible tags
  - $A = a_{11}, a_{12} \ldots a_{n1} \ldots a_{nn}$ # transition prob. matrix
  - $O = \langle o_1, \ldots, o_T \rangle$ # sentence to tag
  - $B = b_i(o_t)$ # prob. of o_t given q_i
    # a.k.a 'emission' probs.

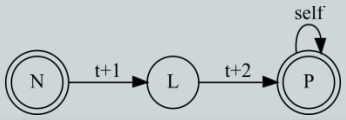


HMMs – formal definition

- 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(VB|TO) = 0.83 (rows give the condition)

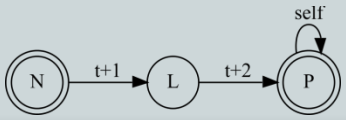


HMMs – formal definition

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

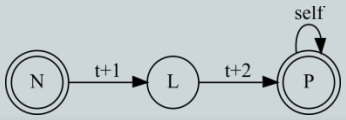
	I	want	to	see
VB	0	0.0093	0	0.12
TO	0	0	0.99	0
NN	0	0.000054	0	0.000007

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



Viterbi code - overview

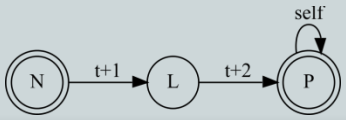
- Prepare emission + transition dictionaries (2 level!)
- Prepare a list to contain each word we see
- Store a dictionary for each word: (N_{tags}*N_{tags} loop)
  - Probability of each prev-tag + tag at this word, based on 3 things:
    - p_so_far * transition_p * emission_p
- Choose best option, store **how we got there** (what was best prev-tag – the backpointer)



Building block: defaultdict

- Uses a **default** value if key is unknown:
  - Should be initializable with a data type or function returning some value

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



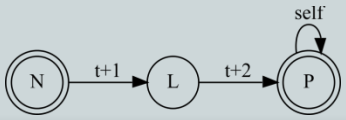
Building block: defaultdict

- 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:

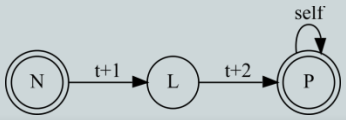
```
# Suppose half_returner is a function, always returns 0.5
def half_returner():
    return 0.5
my_dict = defaultdict(half_returner)
```



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"**] ?

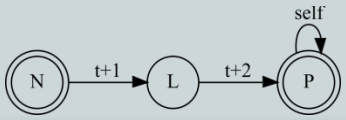


Viterbi - implementation

tagging/viterbi_simple.py

- Imports and states:

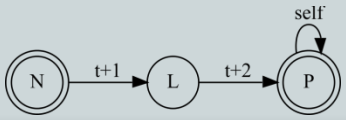
```
from collections import defaultdict


# 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

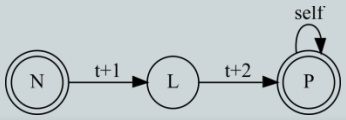
- Training data for transition is a dictionary:

  trans_p ={}

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

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

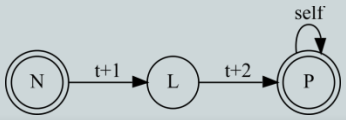
- But these would be regular dictionaries, no default value for missing keys



Viterbi - transitional probabilities

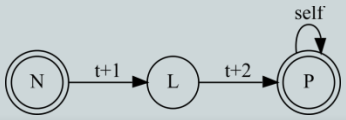
- A better way:

```
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, …}
```



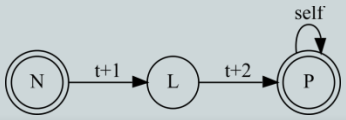
Viterbi – emission probabilities

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

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



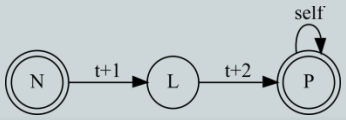
The algorithm 1/3

```
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]]
```



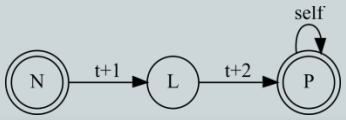
The algorithm 2/3

```
# 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)
```

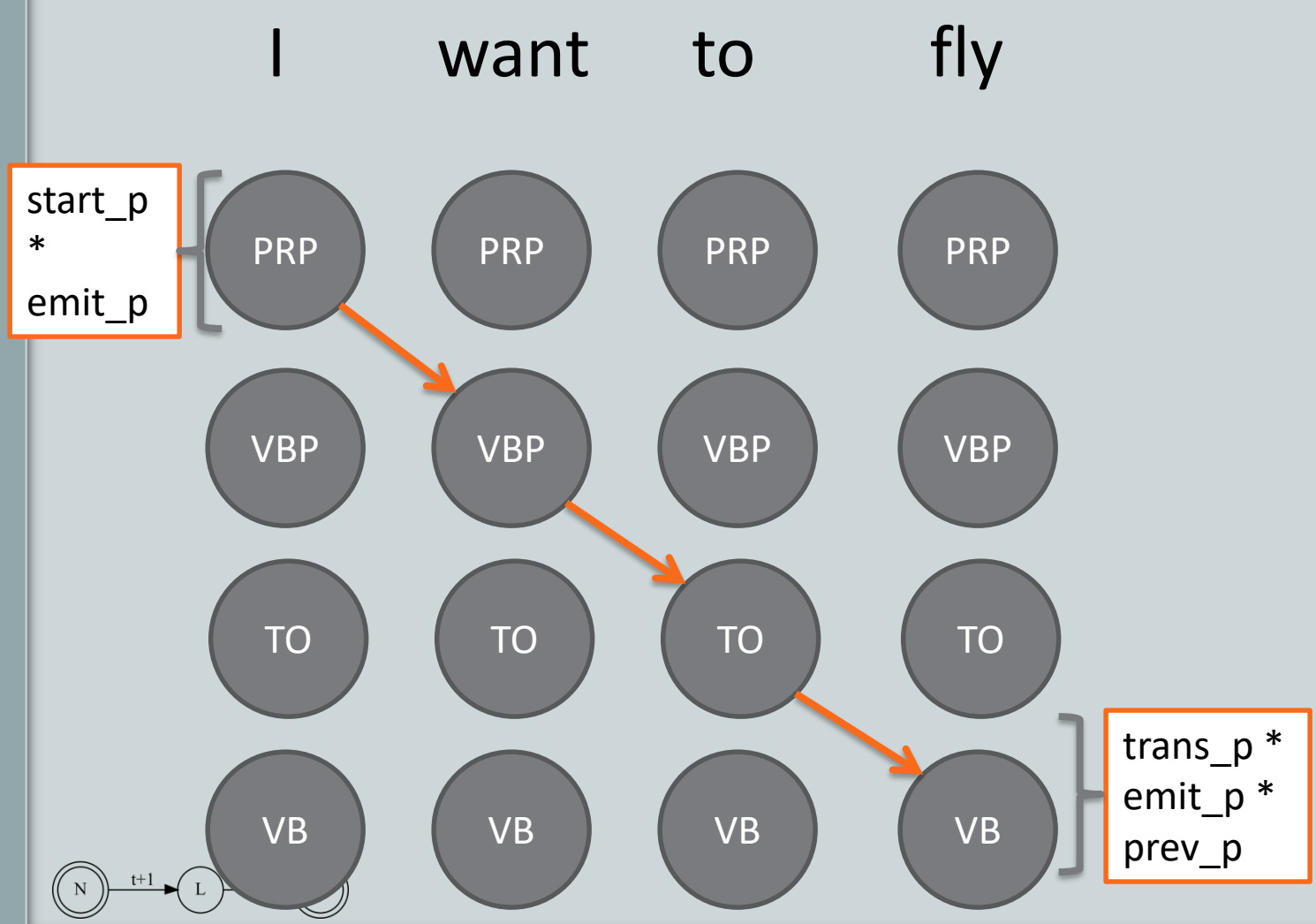


The algorithm 3/3

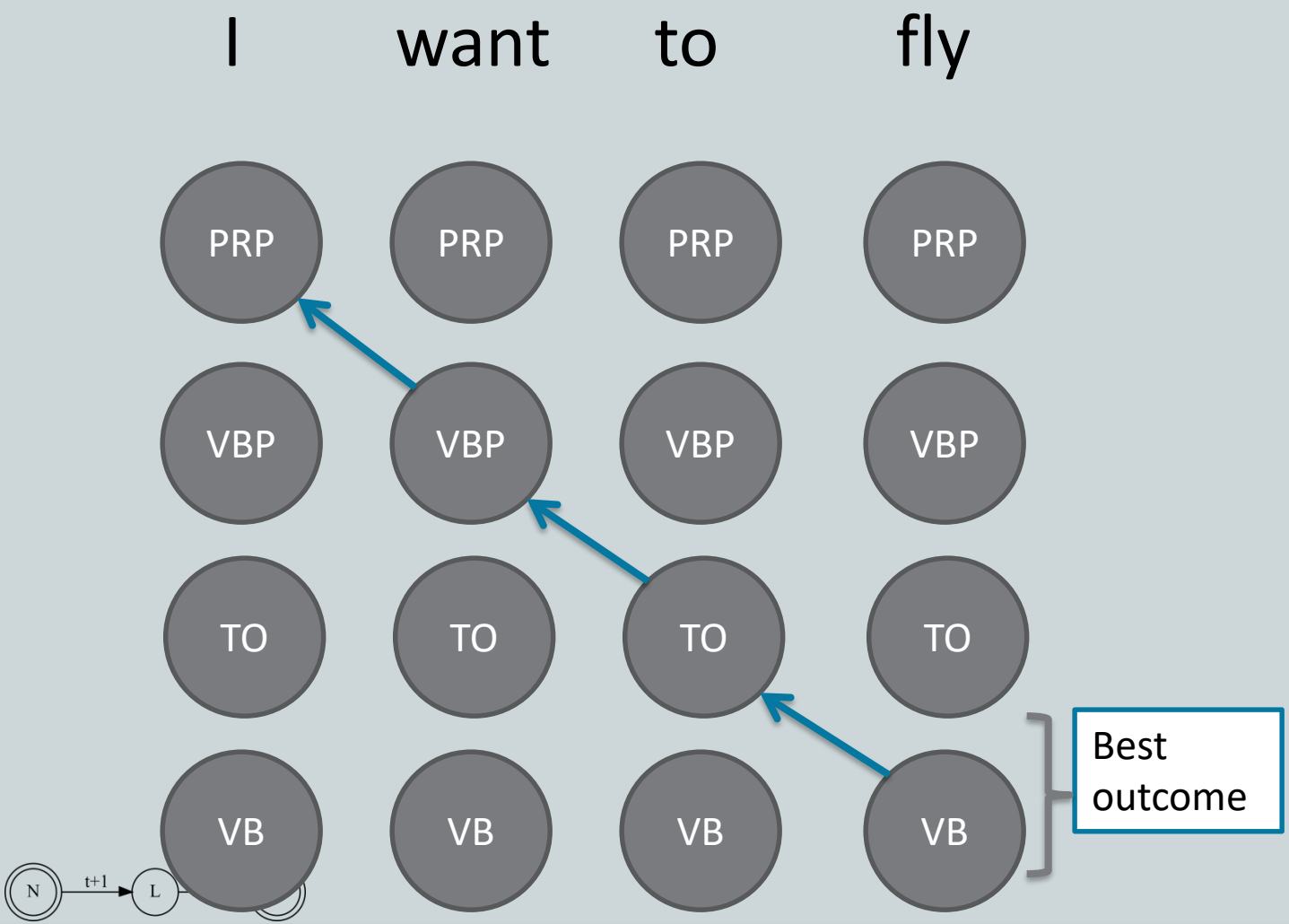
```
optimal_list = []

# 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

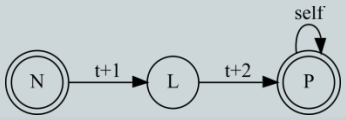


Backtrace step



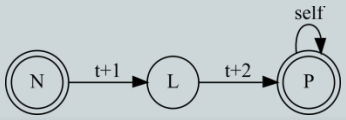
Group work

- 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



From tagging to sequences

- Part of speech labeling is a classic example of token-wise tagging:
  - Input is a sequence of words (tokens)
  - Each word receives exactly one category
  - There are usually no other features except words to decide the correct tag
- But not all labeling tasks are like this!
- We could tag more complex sequences and with more input features!



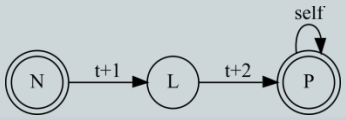
Sequence labeling – NER

- A typical example is Named Entity Recognition
- Not every token is labeled:

PER	**--**		***ORG***	**--**
• Kim	visited		Intel	.

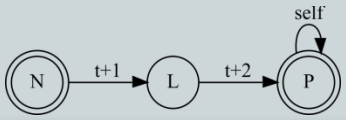
- Labels come in **spans**:

PER	***PER***	**--**		***ORG***	***ORG***
• Kim	Jung	visited		Intel	Corp.



How many spans?

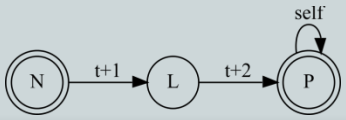
- If we only use labels like PER and ORG, we can treat this as an HMM/Viterbi problem
  - Tags: **PER, ORG, .., --** ('--' is a tag)
  - Input: **Kim, Jung, visited …**
  - Emission probabilities: P(Kim|PER), P(Intel|ORG)
  - Transition probabilities: PER → -- → ORG → ORG
- But how can we tell how many ORGs we have?
  - Intel Corp. ORG ORG → 1 org., 2 tokens
  - IBM Google lawsuit ORG ORG -- → 2 orgs!!



Solution: BIO encoding

- We add more label types to indicate **B**eginning and **I**nside of entities:
  - IBM *B-ORG*
  - Corp. *I-ORG*
  - hired *O*
  - Kim *B-PER*
  - Jung *I-PER*

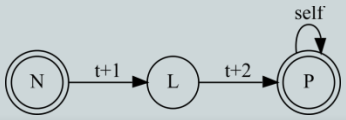
- The label **O** is like our '--': **Outside** any entity



Solution: BIO encoding

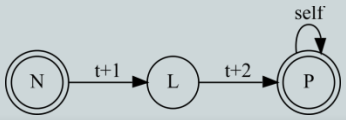
- Labels impose restrictions on transitions:
  - P(B-PER → I-PER) > p(I-PER → B-PER)
  - P(O → I-PER) = 0 (why?)

- We can still use HMM/Viterbi…
- But is just one emission probability enough?
  - P(PER|Kim) …
  - What about other features?



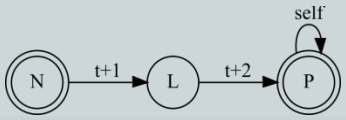
Just one emission?

- Many things influence the probability that a word is a person/company name:
  - Capitalization (very good at finding ‘O’)
  - All caps? (ORG)
  - Word length
  - Knowledge bases (is this in a list of company names? Place names?)
  - …
- Viterbi can’t handle this…



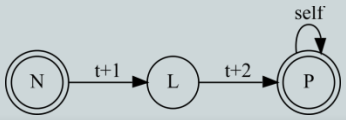
Using multiple features

- Ideally our input should look like this:
  - IBM NNP allcaps … ***B-ORG***
  - Corp. NNP title … ***I-ORG***
  - hired VBD lower … ***O***
  - Kim NNP title … ***B-PER***
  - Jung NNP title … ***I-PER***



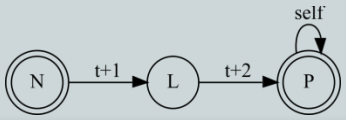
Decoding - CRF

- Efficient decoding over **multiple** features can be done using **Conditional Random Fields** (CRF)
- We do not have time to implement CRF in this course
- For our purposes, a Linear Chain CRF is
  - a sequence label decoder equivalent to a Viterbi decoder
  - using **multiple input features**
  - and **arbitrary functions** for features over the sequence
- Advanced reading: Sutton & McCallum (2006) in Canvas (optional!)



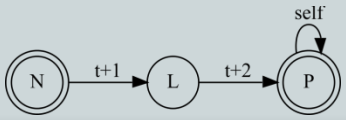
What are the probabilities?

- For smaller datasets, CRF taggers can learn joint discrete feature value distributions:
  - Python library:
    - pip install python-crfsuite (Okazaki 2007)
  - Good off the shelf CRF tagger:
    - Marmot (Müller et al. 2013), http://cistern.cis.lmu.de/marmot/
  - CRF NER tagging example in Canvas:
    - ner/crf_entities.py



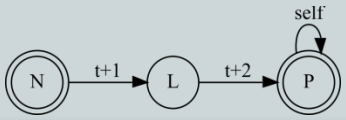
Neural sequence labeling

- Since features can be anything…
- For larger datasets, we can use neural networks
- Word embeddings as features



Popular libraries

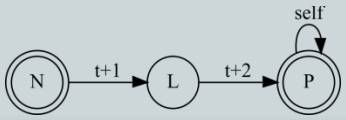
- Flair (Akbik et al. 2019)
- AllenNLP (Gardner et al. 2018)
- NCRF++ (Yang & Zhang 2018)



Example – Flair (Akbik et al. 2019)

```
from flair.models import SequenceTagger

# pretrained NER tagger
tagger = SequenceTagger.load('ner')

sentence = Sentence('George Washington went to Washington .')

# predict NER tags
tagger.predict(sentence)

# print sentence with predicted tags
print(sentence.to_tagged_string())
```

George <B-PER> Washington <E-PER> went to Washington <S-LOC> .



Homework – for Nov 17

- **Turn viterby_simple.py into a trainable tagger!**
  - Split the training filein data/ **en_gum-ud-train.conllu** into a list of lines.
  - For each line that contains a tab ("\t"), split it by tab to collect the word (second column) and PTB part of speech tag (5th column, i.e. [4])
  - Use a dictionary to track frequencies for:
    - Each word as each tag
    - Each transition from the last tag to the next tag
    - Divide by total number of words to make probabilities and put them into the same nested dictionary structure used by the viterby tagger.
  - Bonus: Now read the file **en_gum-ud-dev.conllu** to get test sentences (sentences are separated by blank lines)
    - Collect the words in each sentence from the 2nd column (column [1])
    - Save the correct POS tags from the 5th column as well
    - Modify viterbi() to **return the optimal list of tags**
    - Get tags for each sentence using viterbi and check: for how many **tokens** did the tagger find the right solution?
  - Bonus question: for how many **sentences** is the tagger 100% correct?