0.1 What are Part-of-Speech (POS) Tags?

Syntactic labels of words: This/DET is/VB a/DET simple/ADJ sentence/NOUN. These abstract away from the core word meanings. Sequences (i.e. legal ordering) are mostly (though not entirely) dependent only on the labels, not by the words themselves.

- Open class words ("content words")
  - nouns, verbs, adjectives, adverbs
  - mostly content-bearing. refer to objects, actions, features in the world
  - open class = there is no limit to what they are or can describe so new ones are added all the time (email, website, defenestrate)

- Closed class words ("function words")
  - pronouns, determiners, prepositions, connectives
  - there are a limited number of these
  - mostly functional: to tie the concepts of a sentence together

How many? It depends. Some kind of annotation standard is needed to decide, e.g., should proper and common nouns be separated? Should singular or plural nouns? Present/past/main/aux verbs?

Penn treebank: fairly detailed set of tags (45 of them) used frequently along with parsing (particularly constituent parsing). Some weird quibbles: why does ‘to’ get its own tag?
Morphologically rich languages will often have ‘morphosyntactic’ tags = detailed breakdown of how the word parts combine, such as ‘Noun+A3sg+P2sg+Nom’ with possibly thousands of possibilities. Here’s a Turkish example from Dilek Hakkani-Tur, via Jurafsky and Martin:

Yerdeki izin temizlenmesi gerek iz +Noun+A3Sg+Pnon+Gen

The trace on the floor should be cleaned

A3Sg = 3rd person agreement; Pnon = no possessive agreement, Gen = genitive case

Universal dependencies: 17 tags to try to catch phenomena that are distinct across all languages (there is also a 12 tag variant):

- ADJ adjective
- ADP adposition
- ADV adverb
- AUX auxiliary
- CCONJ coordinating conjunction
- DET determiner
- INTJ interjection
- NOUN noun
- NUM numeral
- PART particle
- PRON pronoun
- PROPN proper noun
- PUNCT punctuation
- SCONJ subordinating conjunction
- SYM symbol
- VERB verb
- X other

0.2 Why is it hard?

Why is it always hard? Ambiguity.

glass of water/NOUN vs water/VERB the plants
lie/VERB down vs tell a lie/NOUN
wind/VERB down vs a mighty wind/NOUN (note these last are homographs)
time flies like an arrow
NOUN VERB MODAL DET NOUN
VERB NOUN
ADJ NOUN VERB
What knowledge do we need?
1) A component deciding based on the word itself (some words only nouns, like arrow, some words ambiguous, a priori tag probability)
2) A component deciding based on the tags of surrounding words (a sequence of two determiners is rare, as is two base form verbs. Determiner almost always followed by adjective or nouns
Could we just put this into a linear model, predicting one tag at a time?

0.3 Why do we care?
This is the first step toward full-sentence syntax (structure trees). POS tags can also be used
as input to tasks people do care about (sentiment analysis, word sense disambiguation).

In neural network land, could these be inferred automatically as ‘features’? With sufficient data, it seems they can, yes. But often there isn’t enough data, and linguistic intuition
guides the search space, so that we can start off with prior knowledge of meaningful relationships between words.

More importantly, beyond text classification, this is a sequence labeling task, and the
approaches we learn here, are specifically helpful for this and other sequence labeling tasks,
of which here are two:

• Named Entity Recognition (NER): label words as beginning to persons (PER), organiza-
tions (ORG), locations (LOC), or none of the above: Barack/PER Obama/PER spoke/N from/N the/N White/LOC House/LOC today/N ./N

• Information field segmentation: Given specific text type (e.g. classified ad), find which
words belong to which "fields" for db creation (price/size/location, author/title/year): 3BR/SIZE apt/TYPE in/N West/LOC Adams/LOC ./N near/LOC USC/LOC ./N Bright/FEAT ./N well/FEAT maintained/FEAT ...

Key features of sequence labeling: Correct label depends on

• item to be labeled (NER: Smith is probably a person. POS: chair is probably a noun)

• labels of surrounding items (NER: if following word is an organization (e.g. Corp.),
then this word is more likely to be an organization. POS: if preceding word is a modal
verb (e.g. will), then this word is more likely to be a verb)

The Hidden Markov Model (HMM) combines these sources probabilistically.
0.4 Bayes’ Law and Assumptions again

For word sequence \( W = w_1, \ldots, w_n \) we want the most likely tag sequence \( T = t_1, \ldots, t_n \), i.e.

\[
\arg\max_T P(T|W) = \arg\max_T P(W|T)P(T)
\]

Then we make some simplifying assumptions:

1) \( P(W|T) = P(w_1, \ldots, w_n|t_1, \ldots, t_n) = P(w_1|w_2, \ldots, w_n, t_1, \ldots, t_n)P(w_2|\ldots)P(w_n|t_1, \ldots, t_n) \approx P(w_1|t_1)P(w_2|t_2)\ldots P(w_n|t_n) \)

In other words we assume independence of all words and tags except \( w_i \) and \( t_i \).

2) \( P(T) = P(t_1, \ldots, t_n) = P(t_n|t_{n-1}, \ldots, t_1)P(t_{n-1}|t_{n-2}, \ldots, t_1) \approx P(t_n|t_{n-1})P(t_{n-1}|t_{n-2})\ldots P(t_2|t_1)P(t_1) \)

In other words we assume a tag is conditioned only on the previous tag. This is called the ‘Markov’ assumption.

We call \( P(W|T) \) the emission probability and \( P(T) \) the transition probability. A plate diagram helps (this is fig. 7.2 of Eisenstein):

Small implementation note: there’s no specific conditioning on the beginning or ending of a sentence, but it does seem like a good kind of probability to have. So in practice we can imagine sentences all beginning with, e.g. ‘BOS’ and ending with ‘EOS’. Note the probability of the first tag is \( P(t_1) \) with no conditioning, but since the first tag is always the same (and the same word is always drawn from it) we don’t need to include those probabilities. It is, however, helpful to know \( P(t_2|BOS) \) and to include a proper draw for ending, i.e. \( P(EOS|t_n) \).

These probabilities are generally learned empirically and then smoothed (it’s much more important to smooth the emission probabilities...why?). Here are tables of empirically learned probabilities (these are from Jurafsky and Martin):

<table>
<thead>
<tr>
<th>( t_{n-1}\backslash t_1 )</th>
<th>NNP</th>
<th>MD</th>
<th>VB</th>
<th>JJ</th>
<th>NN</th>
<th>. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;s&gt;</td>
<td>0.2767</td>
<td>0.0006</td>
<td>0.0031</td>
<td>0.0453</td>
<td>0.0449</td>
<td>. . .</td>
</tr>
<tr>
<td>NNP</td>
<td>0.3777</td>
<td>0.0110</td>
<td>0.0009</td>
<td>0.0084</td>
<td>0.0584</td>
<td>. . .</td>
</tr>
<tr>
<td>MD</td>
<td>0.0008</td>
<td>0.0002</td>
<td>0.7968</td>
<td>0.0005</td>
<td>0.0008</td>
<td>. . .</td>
</tr>
<tr>
<td>VB</td>
<td>0.0322</td>
<td>0.0005</td>
<td>0.0050</td>
<td>0.0837</td>
<td>0.0615</td>
<td>. . .</td>
</tr>
<tr>
<td>JJ</td>
<td>0.0306</td>
<td>0.0004</td>
<td>0.0001</td>
<td>0.0733</td>
<td>0.4509</td>
<td>. . .</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( t_1\backslash w_i )</th>
<th>Janet</th>
<th>will</th>
<th>back</th>
<th>the</th>
<th>. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>NNP</td>
<td>0.000032</td>
<td>0</td>
<td>0</td>
<td>0.000048</td>
<td>. . .</td>
</tr>
<tr>
<td>MD</td>
<td>0</td>
<td>0.308431</td>
<td>0</td>
<td>0</td>
<td>. . .</td>
</tr>
<tr>
<td>VB</td>
<td>0</td>
<td>0.00028</td>
<td>0.000672</td>
<td>0</td>
<td>. . .</td>
</tr>
<tr>
<td>DT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.506099</td>
<td>. . .</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>
So here are the probabilities that would be used to calculate the probability of the tagged sentence Time/NOUN flies/VERB like/MODAL an/DET arrow/NOUN:

\[
P(\text{BOS} | \text{Time}) P(\text{flies} | \text{V}) P(\text{like} | \text{M}) P(\text{an} | \text{D}) P(\text{arrow} | \text{N})
\]

0.5 Viterbi Search

Things become tricky when trying to tag a new sequence; we want, remember, \(\arg\max_T P(W, T)\) but finding this requires searching over the exponential number of sequences; even for a 5 word sequence over the 17-tag UD set, that’s \(17^5 = 1,419,857\) sequences. We turn instead to the Viterbi algorithm, a dynamic programming algorithm which uses the following intuition:

If we are at word \(n - 1\) and have already calculated the best tag sequence ending in each of the 17 tag types at word \(n - 1\), then the best tag sequence to word \(n\) (which has tag EOS) is one of 17 options: the best tag sequence to word \(n - 1\) ending in tag 1 times the score for tag 1 to EOS, etc.

But the best tag sequence to tag 1 at word \(n - 1\) is one of 17 options: the best tag sequence to \(n - 2\) ending in tag 1 times the score for transition from tag 1 to tag 1 (times the emission for word \(n - 1\) with tag 1), etc.

To make things complete, we assume all sequences start with a special BOS word that has an emission probability of 1 for the BOS tag.

It’s helpful to work through the following chart using the provided statistics; we’ll do this in class (note that in code you probably want to add logs instead of multiplying probs):

```python
for t in range(tagset):
    score[t][0] = emis[BOS][t]
for i, w in enumerate(words[1:]):
    for t in range(tagset):
        score[t][i] = max(score[:, i-1]*trans[t, :]*emis[w][t])
```

---

<table>
<thead>
<tr>
<th>trans</th>
<th>N</th>
<th>V</th>
<th>D</th>
<th>P</th>
<th>A</th>
<th>EOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOS</td>
<td>.3</td>
<td>.1</td>
<td>.3</td>
<td>.2</td>
<td>.1</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>.2</td>
<td>.4</td>
<td>.01</td>
<td>.3</td>
<td>.04</td>
<td>.05</td>
</tr>
<tr>
<td>V</td>
<td>.3</td>
<td>.05</td>
<td>.3</td>
<td>.2</td>
<td>.1</td>
<td>.05</td>
</tr>
<tr>
<td>D</td>
<td>.9</td>
<td>.01</td>
<td>.01</td>
<td>.01</td>
<td>.07</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>.4</td>
<td>.05</td>
<td>.4</td>
<td>.1</td>
<td>.05</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>.1</td>
<td>.5</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
<td>.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>emis</th>
<th>BOS</th>
<th>a</th>
<th>cat</th>
<th>doctor</th>
<th>in</th>
<th>is</th>
<th>the</th>
<th>very</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOS</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>0</td>
<td>0</td>
<td>.5</td>
<td>.4</td>
<td>0</td>
<td>.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V</td>
<td>0</td>
<td>0</td>
<td>.1</td>
<td>0</td>
<td>.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>.7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>.1</td>
<td>0</td>
<td>0</td>
<td>.9</td>
</tr>
</tbody>
</table>

5
0.6 OK, but what about more interesting features and discriminative models?

In fact, we can use the perceptron algorithm here, just as we used it for simple whole-text label prediction. Let’s revisit perceptron:
def train(labeled_sentences, options, featsize):
    model = {'theta': np.random(featsize)}
    for i in range(iterations): # user-determined
        for sentence, label in labeled_sentences:
            hyp = classify(sentence, options, model)
            if hyp != label:
                model['theta'] += feats(sentence, label)−feats(sentence, hyp)
    return model

def classify(sentence, options, model):
    scores = {}
    for option in options:
        scores[option] = evaluate(sentence, option, model)
    return max(scores, key=scores.get)

The real problem is the classify function, which I included here, or actually the options set. The point of classify is to find the maximum scoring sequence over all options. In sentiment classification there were only three sentiments, so you could just do 3 lookups and choose the best. But if there are t possible tags and the sequence is n words long, there are $t^n$ possible sequences. Thankfully, dynamic programming in general (and the Viterbi algorithm specifically) lets us efficiently find the maximum score sequence. This then becomes what is known as structured perceptron since you are ultimately learning to predict the structured sequence of labels, and not just a single label at a time. We are no longer constrained to the emission and transition probabilities.

Here is the Viterbi algorithm, rewritten to use general features, assuming an appropriate current $\theta$ and a feats function which returns features based on the last label $t'$, the current word $w$, and the proposed current label $t$ (column vector notation won’t work here but the complexity hasn’t increased despite the deeper nesting):

for t in range(tagset):
    score[t][0] = theta*feats(emptyset, BOS, t)
for i, w in enumerate(words[1:]):
    for t in range(tagset):
        score[t][i] = −infty
    for s in range(tagset):
        score[t][i] = max(score[t][i], score[s][i−1]*theta*feats(s, w, t))

What about logistic regression? Yes, this can be done as well (logistic regression over structures is called conditional random fields and the specific dependencies we allow make this a linear chain conditional random field), but it again requires efficiency changes, now to both inference and learning. For inference we have already seen how to efficiently calculate; the Viterbi algorithm is used to determine the argmax sequence. Let the model score of a label sequence $y$ given word sequence $w$ be $\psi(w, y)$. Then the conditional probability we seek is:

$$p(y|w) = \frac{\exp(\psi(w, y))}{\sum_{y'} \exp(\psi(w, y'))}$$
and the loss is $-\log p(y|w)$.\footnote{Eisenstein adds a $L_2$ norm which is a distraction.} It should be clear that for learning, we need to calculate the sum of all scores (that denominator of softmax); this is an important component in gradient calculation. This can be calculated with a slight variant to the Viterbi algorithm called the Forward algorithm.

Compare the Forward algorithm to the previous Viterbi algorithm:

```python
for t in range(tagset):
    score[t][0] = theta*feats(emptyset, BOS, t)
for i, w in enumerate(words[1:]):
    for t in range(tagset):
        score[t][i] = 0
    for s in range(tagset):
        score[t][i] += score[s][i-1]*theta*feats(s, w, t)
```

This calculates the partial score of all sequence label prefixes from word 0 to $t$. The analogous Backward algorithm does the same calculation, but starts at the end of the sequence:

```python
for t in range(tagset):
    score[t][n] = 1
for i, w in reversed(enumerate(words)):
    for s in range(tagset):
        score[s][i] = 0
    for t in range(tagset):
        score[s][i] += score[t][i+1]*theta*feats(s, w, t)
```

Rather than directly calculate features, naturally, we can use neural networks, and this is in practice what is done today. However, the issue of exponential search still applies. Features are calculated via a bidirectional RNN and those features are provided to a CRF. The whole thing is differentiable and can be learned jointly (and is implemented using frameworks like PyTorch). The set of features for each possibility is also calculatable via the same dynamic programming approach used to calculate Forward/Viterbi.