

Natural Language Processing

# Sequence labeling



# 8 Parts of Speech

## NOUN

A **noun** names a person, place, things or idea.

**Examples**

*dog, cat, horse, student, teacher, apple, Mary and etc...*

## VERB

A **verb** is a word or group of words that describes an action, experience.

**Examples**

*realize, walk, see, look, sing, sit, listen and etc...*

## PREPOSITION

A **preposition** is used before a noun, pronoun, or gerund to show place, time, direction in a sentence.

**Examples**

*at, in, on, about, to, for, from and etc...*

## PRONOUN

**Pronouns** replace the name of a person, place, thing or idea in a sentence.

**Examples**

*he, she, it, we, they, him, her, this ,that and etc...*

## ADVERB

An **adverb** tells how often, how, when, where. It can describe a verb, an adjective or an adverb.

**Examples**

*loudly, always, never, late, soon etc...*

## ADJECTIVE

An **adjective** describes a noun or pronoun.

**Examples;** red, tall, fat, long, short, blue, beautiful, sour, bitter and etc...

## CONJUNCTION

**Conjunctions** join words or groups of words in a sentence.

**Examples;** and, because, yet, therefore, moreover, since, or, so, until, but and etc...

## INTERJECTION

**Interjections** express strong emotion and is often followed by an exclamation point.

**Examples**

*Bravo! Well! Aha! Hooray! Yeah! Oops! Phew!*

Proper names are another important and anciently studied linguistic category. While parts of speech are generally assigned to individual words or morphemes, a proper name is often an entire multiword phrase, like the name “Marie Curie”, the location “New York City”, or the organization “Stanford University”. We’ll use the term named entity for, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization, although as we’ll see the term is commonly extended to include things that aren’t entities per se.

Parts of speech (also known as POS) and named entities are useful clues to sentence structure and meaning. Knowing whether a word is a noun or a verb tells us about likely neighboring words (nouns in English are preceded by determiners and adjectives, verbs by nouns) and syntactic structure (verbs have dependency links to nouns), making part-of-speech tagging a key aspect of parsing. Knowing if a named entity like Washington is a name of a person, a place, or a university is important to many natural language processing tasks like question answering, stance detection, or information extraction.

In this chapter we’ll introduce the task of part-of-speech tagging, taking a sequence of words and assigning each word a part of speech like NOUN or VERB, and the task of named entity recognition (NER), assigning words or phrases tags like PERSON, LOCATION, or ORGANIZATION.

Such tasks in which we assign, to each word  $x_i$  in an input word sequence, a label  $y_i$ , so that the output sequence  $Y$  has the same length as the input sequence  $X$  are called sequence labeling tasks. We’ll introduce classic sequence labeling algorithms, one generative— the Hidden Markov Model (HMM)— and one discriminative— the Conditional Random Field (CRF).

In following chapters we’ll introduce modern sequence labelers based on RNNs and Transformers.

Tag Description	Example	Tag	Description	Example	Tag	Description	Example
CC coord. conj.	<i>and, but, or</i>	NNP	proper noun, sing.	<i>IBM</i>	TO	"to"	<i>to</i>
CD cardinal number	<i>one, two</i>	NNPS	proper noun, plu.	<i>Carolinas</i>	UH	interjection	<i>ah, oops</i>
DT determiner	<i>a, the</i>	NNS	noun, plural	<i>llamas</i>	VB	verb base	<i>eat</i>
EX existential 'there'	<i>there</i>	PDT	predeterminer	<i>all, both</i>	VBD	verb past tense	<i>ate</i>
FW foreign word	<i>mea culpa</i>	POS	possessive ending	's	VBG	verb gerund	<i>eating</i>
IN preposition/ subordin-conj	<i>of, in, by</i>	PRP	personal pronoun	<i>I, you, he</i>	VBN	verb past participle	<i>eaten</i>
JJ adjective	<i>yellow</i>	PRP\$	possess. pronoun	<i>your, one's</i>	VBP	verb non-3sg-pr	<i>eat</i>
JJR comparative adj	<i>bigger</i>	RB	adverb	<i>quickly</i>	VBZ	verb 3sg pres	<i>eats</i>
JJS superlative adj	<i>wildest</i>	RBR	comparative adv	<i>faster</i>	WDT	wh-determ.	<i>which, that</i>
LS list item marker	<i>1, 2, One</i>	RBS	superlatv. adv	<i>fastest</i>	WP	wh-pronoun	<i>what, who</i>
MD modal	<i>can, should</i>	RP	particle	<i>up, off</i>	WP\$	wh-possess.	<i>whose</i>
NN sing or mass noun	<i>llama</i>	SYM	symbol	+,%,&	WRB	wh-adverb	<i>how, where</i>

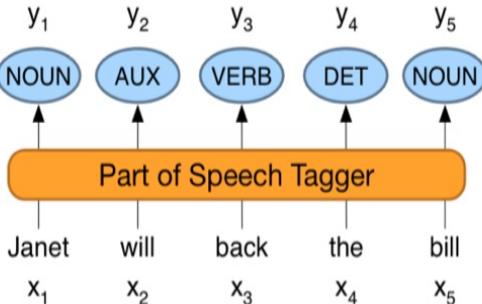
Figure 8.2 Penn Treebank part-of-speech tags.

Part-of-speech tagging is the process of assigning a part-of-speech to each word in a text. The input is a sequence  $x_1, x_2, \dots, x_n$  of (tokenized) words and a tagset, and the output is a sequence  $y_1, y_2, \dots, y_n$  of tags, each output  $y_i$  corresponding exactly to one input  $x_i$ , as shown in the intuition in Fig. 8.3.

Tagging is a disambiguation task; words are ambiguous — have more than one possible part-of-speech—and the goal is to find the correct tag for the situation. For example, *book* can be a verb (*book that flight*) or a noun (*hand me that book*). That can be a determiner (*Does that flight serve dinner*) or a complementizer (*I thought that your flight was earlier*). The goal of POS-tagging is to resolve these ambiguities, choosing the proper tag for the context.

The accuracy of part-of-speech tagging algorithms (the percentage of test set tags that match human gold labels) is extremely high. One study found accuracies over 97% across 15 languages from the Universal Dependency (UD) treebank (Wu and Dredze, 2019).

Accuracies on various English treebanks are also 97% (no matter the algorithm; HMMs, CRFs, BERT perform similarly). This 97% number is also about the human performance on this task, at least for English (Manning, 2011).



**Figure 8.3** The task of part-of-speech tagging: mapping from input words  $x_1, x_2, \dots, x_n$  to output POS tags  $y_1, y_2, \dots, y_n$ .

Disambiguation (resolving ambiguity) of POS tags:

Probabilistic approach: to find the best POS tag sequence of all possible sequences by using a conditional probability (scoring).

$$\hat{y} = \text{argmax } P(y|x)$$

$\hat{y}$  means "our estimation for  $y^n$ "

argmax: to find  $y$  that maximizes  $P(y|x)$

How to model this?

→ HMM / Structured perceptron / Conditional Random Fields (CRFs) etc.

→ For training: ML / MAP / SGD

→ Inference: how to compute argmax  $P(y|x)$  efficiently using Viterbi algorithm.

## 8.3 Named Entities and Named Entity Tagging

Part of speech tagging can tell us that words like *Janet*, *Stanford University*, and *Colorado* are all proper nouns; being a proper noun is a grammatical property of these words. But viewed from a semantic perspective, these proper nouns refer to different kinds of entities: *Janet* is a person, Stanford University is an organization, and Colorado is a location.

named entity

named entity  
recognition

NER

A **named entity** is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. The task of **named entity recognition (NER)** is to find spans of text that constitute proper names and tag the type of the entity. Four entity tags are most common: **PER** (person), **LOC** (location), **ORG** (organization), or **GPE** (geo-political entity). However, the term **named entity** is commonly extended to include things that aren't entities per se, including dates, times, and other kinds of temporal expressions, and even numerical expressions like prices. Here's an example of the output of an NER tagger:

PER (person)  
LOC (location)  
ORG (organization)  
GPE (geo-political entity)

Citing high fuel prices, [ORG United Airlines] said [TIME Friday] it has increased fares by [MONEY \$6] per round trip on flights to some cities also served by lower-cost carriers. [ORG American Airlines], a unit of [ORG AMR Corp.], immediately matched the move, spokesman [PER Tim Wagner] said. [ORG United], a unit of [ORG UAL Corp.], said the increase took effect [TIME Thursday] and applies to most routes where it competes against discount carriers, such as [LOC Chicago] to [LOC Dallas] and [LOC Denver] to [LOC San Francisco].

The text contains 13 mentions of named entities including 5 organizations, 4 locations, 2 times, 1 person, and 1 mention of money. Figure 8.5 shows typical generic named entity types. Many applications will also need to use specific entity types like proteins, genes, commercial products, or works of art.

Type	Tag	Sample Categories	Example sentences
People	PER	people, characters	Turing is a giant of computer science.
Organization	ORG	companies, sports teams	The IPCC warned about the cyclone.
Location	LOC	regions, mountains, seas	Mt. Sanitas is in Sunshine Canyon.
Geo-Political Entity	GPE	countries, states	Palo Alto is raising the fees for parking.

Figure 8.5 A list of generic named entity types with the kinds of entities they refer to.

Named entity tagging is a useful first step in lots of natural language processing tasks. In sentiment analysis we might want to know a consumer's sentiment toward a particular entity. Entities are a useful first stage in question answering, or for linking text to information in structured knowledge sources like Wikipedia. And named entity tagging is also central to tasks involving building semantic representations, like extracting events and the relationship between participants.

Unlike part-of-speech tagging, where there is no segmentation problem since each word gets one tag, the task of named entity recognition is to find and label spans of text, and is difficult partly because of the ambiguity of segmentation; we

<sup>1</sup> In English, on the WSJ corpus, tested on sections 22–24.

need to decide what's an entity and what isn't, and where the boundaries are. Indeed, most words in a text will not be named entities. Another difficulty is caused by type ambiguity. The mention JFK can refer to a person, the airport in New York, or any number of schools, bridges, and streets around the United States. Some examples of this kind of cross-type confusion are given in Figure 8.6.

[PER Washington] was born into slavery on the farm of James Burroughs.

[ORG Washington] went up 2 games to 1 in the four-game series.

Blair arrived in [LOC Washington] for what may well be his last state visit.

In June, [GPE Washington] passed a primary seatbelt law.

**Figure 8.6** Examples of type ambiguities in the use of the name *Washington*.

The standard approach to sequence labeling for a span-recognition problem like NER is **BIO** tagging (Ramshaw and Marcus, 1995). This is a method that allows us to treat NER like a word-by-word sequence labeling task, via tags that capture both the boundary and the named entity type. Consider the following sentence:

[PER Jane Villanueva] of [ORG United], a unit of [ORG United Airlines Holding], said the fare applies to the [LOC Chicago] route.

Figure 8.7 shows the same excerpt represented with **BIO** tagging, as well as variants called **IO** tagging and **BIOES** tagging. In BIO tagging we label any token that begins a span of interest with the label B, tokens that occur inside a span are tagged with an I, and any tokens outside of any span of interest are labeled O. While there is only one O tag, we'll have distinct B and I tags for each named entity class. The number of tags is thus  $2n + 1$  tags, where  $n$  is the number of entity types. BIO tagging can represent exactly the same information as the bracketed notation, but has the advantage that we can represent the task in the same simple sequence modeling way as part-of-speech tagging: assigning a single label  $y_i$  to each input word  $x_i$ :

Words	IO Label	BIO Label	BIOES Label
Jane	I-PER	B-PER	B-PER
Villanueva	I-PER	I-PER	E-PER
of	O	O	O
United	I-ORG	B-ORG	B-ORG
Airlines	I-ORG	I-ORG	I-ORG
Holding	I-ORG	I-ORG	E-ORG
discussed	O	O	O
the	O	O	O
Chicago	I-LOC	B-LOC	S-LOC
route	O	O	O
.	O	O	O

**Figure 8.7** NER as a sequence model, showing IO, BIO, and BIOES taggings.

We've also shown two variant tagging schemes: IO tagging, which loses some information by eliminating the B tag, and BIOES tagging, which adds an end tag E for the end of a span, and a span tag S for a span consisting of only one word. A sequence labeler (HMM, CRF, RNN, Transformer, etc.) is trained to label each token in a text with tags that indicate the presence (or absence) of particular kinds of named entities.

## 8.4 HMM Part-of-Speech Tagging

In this section we introduce our first sequence labeling algorithm, the Hidden Markov Model, and show how to apply it to part-of-speech tagging. Recall that a sequence labeler is a model whose job is to assign a label to each unit in a sequence, thus mapping a sequence of observations to a sequence of labels of the same length. The HMM is a classic model that introduces many of the key concepts of sequence modeling that we will see again in more modern models.

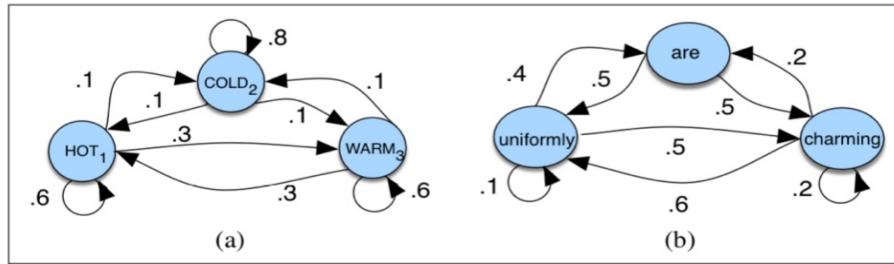
An HMM is a probabilistic sequence model: given a sequence of units (words, letters, morphemes, sentences, whatever), it computes a probability distribution over possible sequences of labels and chooses the best label sequence.

OK

### 8.4.1 Markov Chains

#### Markov chain

The HMM is based on augmenting the Markov chain. A **Markov chain** is a model that tells us something about the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, for example the weather. A Markov chain makes a very strong assumption that if we want to predict the future in the sequence, all that matters is the current state. All the states before the current state have no impact on the future except via the current state. It's as if to predict tomorrow's weather you could examine today's weather but you weren't allowed to look at yesterday's weather.



**Figure 8.8** A Markov chain for weather (a) and one for words (b), showing states and transitions. A start distribution  $\pi$  is required; setting  $\pi = [0.1, 0.7, 0.2]$  for (a) would mean a probability 0.7 of starting in state 2 (cold), probability 0.1 of starting in state 1 (hot), etc.

#### Markov assumption

More formally, consider a sequence of state variables  $q_1, q_2, \dots, q_i$ . A Markov model embodies the **Markov assumption** on the probabilities of this sequence: that when predicting the future, the past doesn't matter, only the present.

$$\text{Markov Assumption: } P(q_i = a | q_1 \dots q_{i-1}) = P(q_i = a | q_{i-1}) \quad (8.3)$$

Figure 8.8a shows a Markov chain for assigning a probability to a sequence of weather events, for which the vocabulary consists of HOT, COLD, and WARM. The states are represented as nodes in the graph, and the transitions, with their probabilities, as edges. The transitions are probabilities: the values of arcs leaving a given state must sum to 1. Figure 8.8b shows a Markov chain for assigning a probability to a sequence of words  $w_1 \dots w_t$ . This Markov chain should be familiar; in fact, it represents a bigram language model, with each edge expressing the probability  $p(w_i | w_j)$ ! Given the two models in Fig. 8.8, we can assign a probability to any sequence from our vocabulary.

Formally, a Markov chain is specified by the following components:

$$Q = q_1 q_2 \dots q_N$$

a set of  $N$  states

$$A = a_{11} a_{12} \dots a_{N1} \dots a_{NN}$$

a **transition probability matrix**  $A$ , each  $a_{ij}$  representing the probability of moving from state  $i$  to state  $j$ , s.t.  $\sum_{j=1}^n a_{ij} = 1 \quad \forall i$

$$\pi = \pi_1, \pi_2, \dots, \pi_N$$

an **initial probability distribution** over states.  $\pi_i$  is the probability that the Markov chain will start in state  $i$ . Some states  $j$  may have  $\pi_j = 0$ , meaning that they cannot be initial states. Also,  $\sum_{i=1}^n \pi_i = 1$

Before you go on, use the sample probabilities in Fig. 8.8a (with  $\pi = [0.1, 0.7, 0.2]$ ) to compute the probability of each of the following sequences:

(8.4) hot hot hot hot

(8.5) cold hot cold hot

What does the difference in these probabilities tell you about a real-world weather fact encoded in Fig. 8.8a?

### 8.4.2 The Hidden Markov Model

A Markov chain is useful when we need to compute a probability for a sequence of observable events. In many cases, however, the events we are interested in are **hidden**: we don't observe them directly. For example we don't normally observe part-of-speech tags in a text. Rather, we see words, and must infer the tags from the word sequence. We call the tags **hidden** because they are not observed.

A **hidden Markov model** (HMM) allows us to talk about both *observed* events (like words that we see in the input) and *hidden* events (like part-of-speech tags) that we think of as causal factors in our probabilistic model. An HMM is specified by the following components:

$$Q = q_1 q_2 \dots q_N$$

a set of  $N$  states

$$A = a_{11} \dots a_{ij} \dots a_{NN}$$

a **transition probability matrix**  $A$ , each  $a_{ij}$  representing the probability of moving from state  $i$  to state  $j$ , s.t.  $\sum_{j=1}^N a_{ij} = 1 \quad \forall i$

$$O = o_1 o_2 \dots o_T$$

a sequence of  $T$  **observations**, each one drawn from a vocabulary  $V = v_1, v_2, \dots, v_V$

$$B = b_i(o_t)$$

a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation  $o_t$  being generated from a state  $q_i$

$$\pi = \pi_1, \pi_2, \dots, \pi_N$$

an **initial probability distribution** over states.  $\pi_i$  is the probability that the Markov chain will start in state  $i$ . Some states  $j$  may have  $\pi_j = 0$ , meaning that they cannot be initial states. Also,  $\sum_{i=1}^n \pi_i = 1$

A first-order hidden Markov model instantiates two simplifying assumptions.

First, as with a first-order Markov chain, the probability of a particular state depends only on the previous state:

$$\boxed{\text{Markov Assumption: } P(q_i|q_1, \dots, q_{i-1}) = P(q_i|q_{i-1})} \quad (8.6)$$

Second, the probability of an output observation  $o_i$  depends only on the state that produced the observation  $q_i$  and not on any other states or any other observations:

$$\boxed{\text{Output Independence: } P(o_i|q_1, \dots, q_i, \dots, q_T, o_1, \dots, o_{i-1}, o_{i+1}, \dots, o_T) = P(o_i|q_i)} \quad (8.7)$$

### 8.4.3 The components of an HMM tagger

Let's start by looking at the pieces of an HMM tagger, and then we'll see how to use it to tag. An HMM has two components, the  $A$  and  $B$  probabilities.

The  $A$  matrix contains the tag transition probabilities  $P(t_i|t_{i-1})$  which represent the probability of a tag occurring given the previous tag. For example, modal verbs like *will* are very likely to be followed by a verb in the base form, a VB, like *race*, so we expect this probability to be high. We compute the maximum likelihood estimate of this transition probability by counting, out of the times we see the first tag in a labeled corpus, how often the first tag is followed by the second:

$$\text{transition probability} \rightarrow P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} \quad (8.8)$$

In the WSJ corpus, for example, MD occurs 13124 times of which it is followed by VB 10471, for an MLE estimate of

$$P(VB|MD) = \frac{C(MD, VB)}{C(MD)} = \frac{10471}{13124} = .80 \quad (8.9)$$

Let's walk through an example, seeing how these probabilities are estimated and used in a sample tagging task, before we return to the algorithm for decoding.

In HMM tagging, the probabilities are estimated by counting on a tagged training corpus. For this example we'll use the tagged WSJ corpus.

The  $B$  emission probabilities,  $P(w_i|t_i)$ , represent the probability, given a tag (say MD), that it will be associated with a given word (say *will*). The MLE of the emission probability is

$$\text{emission probability} \rightarrow P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)} \quad (8.10)$$

Of the 13124 occurrences of MD in the WSJ corpus, it is associated with *will* 4046 times:

$$P(will|MD) = \frac{C(MD, will)}{C(MD)} = \frac{4046}{13124} = .31 \quad \checkmark \quad (8.11)$$

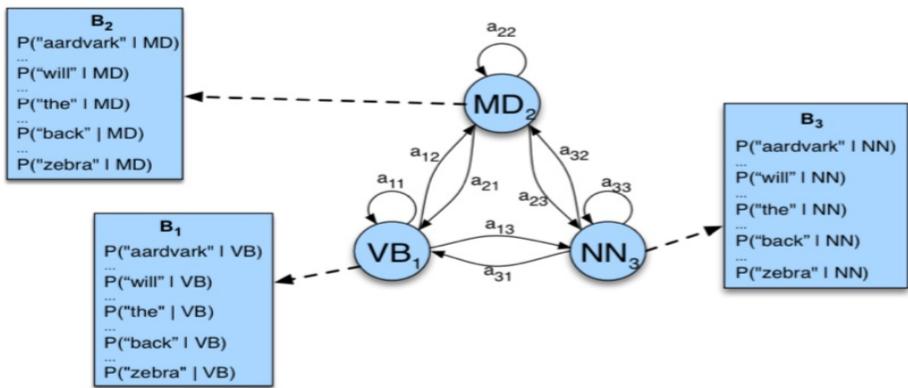
We saw this kind of Bayesian modeling in Chapter 4; recall that this likelihood term is not asking "which is the most likely tag for the word *will*?" That would be the posterior  $P(MD|will)$ . Instead,  $P(will|MD)$  answers the slightly counterintuitive question "If we were going to generate a MD, how likely is it that this modal would be *will*?"

The  $A$  transition probabilities, and  $B$  observation likelihoods of the HMM are illustrated in Fig. 8.9 for three states in an HMM part-of-speech tagger; the full tagger would have one state for each tag.

### 8.4.4 HMM tagging as decoding

For any model, such as an HMM, that contains hidden variables, the task of determining the hidden variables sequence corresponding to the sequence of observations is called **decoding**. More formally,

**Decoding:** Given as input an HMM  $\lambda = (A, B)$  and a sequence of observations  $O = o_1, o_2, \dots, o_T$ , find the most probable sequence of states  $Q = q_1 q_2 q_3 \dots q_T$ .



**Figure 8.9** An illustration of the two parts of an HMM representation: the  $A$  transition probabilities used to compute the prior probability, and the  $B$  observation likelihoods that are associated with each state, one likelihood for each possible observation word.

For part-of-speech tagging, the goal of HMM decoding is to choose the tag sequence  $t_1 \dots t_n$  that is most probable given the observation sequence of  $n$  words  $w_1 \dots w_n$ :

$$\hat{t}_{1:n} = \operatorname{argmax}_{t_1 \dots t_n} P(t_1 \dots t_n | w_1 \dots w_n) \quad (8.12)$$

The way we'll do this in the HMM is to use Bayes' rule to instead compute:

$$\hat{t}_{1:n} = \operatorname{argmax}_{t_1 \dots t_n} \frac{P(w_1 \dots w_n | t_1 \dots t_n) P(t_1 \dots t_n)}{P(w_1 \dots w_n)} \quad (8.13)$$

Furthermore, we simplify Eq. 8.13 by dropping the denominator  $P(w_1^n)$ :

$$\hat{t}_{1:n} = \operatorname{argmax}_{t_1 \dots t_n} P(w_1 \dots w_n | t_1 \dots t_n) P(t_1 \dots t_n) \quad (8.14)$$

HMM taggers make two further simplifying assumptions. The first is that the probability of a word appearing depends only on its own tag and is independent of neighboring words and tags:

$$P(w_1 \dots w_n | t_1 \dots t_n) \approx \prod_{i=1}^n P(w_i | t_i) \quad (8.15)$$

The second assumption, the **bigram** assumption, is that the probability of a tag is dependent only on the previous tag, rather than the entire tag sequence;

$$P(t_1 \dots t_n) \approx \prod_{i=1}^n P(t_i | t_{i-1}) \quad (8.16)$$

Plugging the simplifying assumptions from Eq. 8.15 and Eq. 8.16 into Eq. 8.14 results in the following equation for the most probable tag sequence from a bigram tagger:

$$\hat{t}_{1:n} = \operatorname{argmax}_{t_1 \dots t_n} P(t_1 \dots t_n | w_1 \dots w_n) \approx \operatorname{argmax}_{t_1 \dots t_n} \prod_{i=1}^n \overbrace{P(w_i | t_i)}^{\text{emission transition}} \overbrace{P(t_i | t_{i-1})}^{\text{transition probability}} \quad (8.17)$$

The two parts of Eq. 8.17 correspond neatly to the **B emission probability** and **A transition probability** that we just defined above!

if we go for all words for each tag the the number  
of candidate sequences is  $n^t$  → no of tokens in  
an input sentence.

$t$

It is too huge.

↓  
viterbi algorithm is efficient to  
calculate.

It has Computational cost:  $O(n^2 \cdot t)$  (dynamic  
programming).

## Hidden Markov Models and the Viterbi algorithm

An HMM  $H = (p_{ij}, e_i(a), w_i)$  is understood to have  $N$  hidden Markov states labelled by  $i$  ( $1 \leq i \leq N$ ), and  $M$  possible observables for each state, labelled by  $a$  ( $1 \leq a \leq M$ ). The state transition probabilities are  $p_{ij} = p(q_{t+1} = j | q_t = i)$ ,  $1 \leq i, j \leq N$  (where  $q_t$  is the hidden state at time  $t$ ), the emission probability for the observable  $a$  from state  $i$  is  $e_i(a) = p(O_t = a | q_t = i)$  (where  $O_t$  is the observation at time  $t$ ), and the initial state probabilities are  $w_i = p(q_1 = i)$ .

Given a sequence of observations  $O = O_1 O_2 \cdots O_T$ , and an HMM  $H = (p_{ij}, e_i(a), w_i)$ , we wish to find the maximum probability state path  $Q = q_1 q_2 \cdots q_T$ . This can be done recursively using the Viterbi algorithm.

Let  $v_i(t)$  be the probability of the most probable path ending in state  $i$  at time  $t$ , i.e.,

$$v_i(t) = \max_{q_1, q_2, \dots, q_{t-1}} P(q_1 q_2 \cdots q_{t-1}, q_t = i, O_1 O_2 \cdots O_t | H),$$

and let  $w_i$  be the initial probabilities of the states  $i$  at time  $t = 1$ . (Note this notation avoids the frightening greek letters  $\delta$ ,  $\pi$ , and  $\lambda$  used in the Rabiner notes, using instead  $v$  for Viterbi,  $w$  for weights, and  $H$  for hidden Markov model. The correspondence with the notation used in the Rabiner notes is  $v_i(t) \leftrightarrow \delta_t(i)$ ,  $e_i(a) \leftrightarrow b_i(a)$ ,  $p_{ij} \leftrightarrow a_{ij}$ ,  $w_i \leftrightarrow \pi_i$ ,  $H \leftrightarrow \lambda$ .)

Then  $v_j(t)$  can be calculated recursively using

$$v_j(t) = \max_{1 \leq i \leq N} [v_i(t-1) p_{ij}] e_j(O_t)$$

transition probability  
from state  $i$  to  
state  $j$

previus viterbi path probability

together with initialization

$$v_i(1) = w_i e_i(O_1) \quad 1 \leq i \leq N$$

state observation  
likelihood of observation  
of given the current  
state  $j$ .

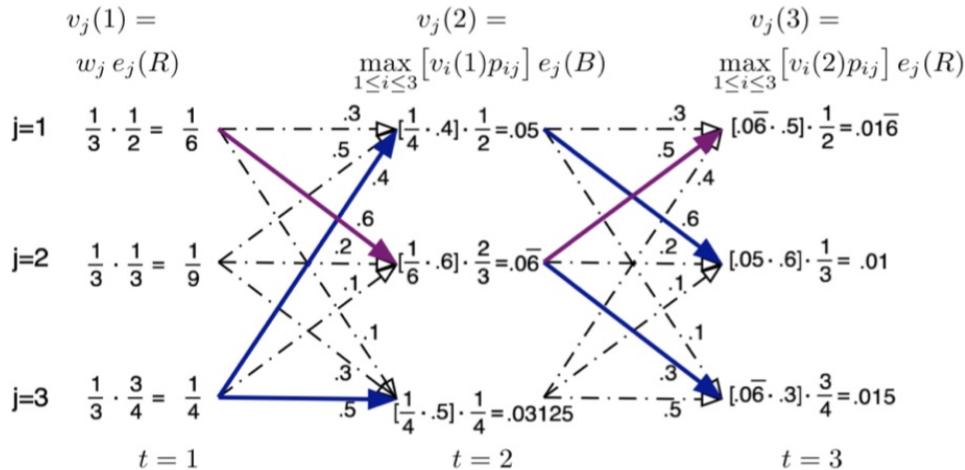
and termination

$$P^* = \max_{1 \leq i \leq N} [v_i(T)]$$

(i.e., at the end we choose the highest probability endpoint, and then we backtrack from there to find the highest probability path).

Note that the maximally likely path is not the only possible optimality criterion, for example choosing the most likely state at any given time requires a different algorithm and can give a slightly different result. But the overall most likely path provided by the Viterbi algorithm provides an optimal state sequence for many purposes.

To illustrate this, consider a three state HMM, with  $R$  or  $B$  emitted by each state (e.g., three urns, each with red or blue balls) with emission probabilities  $e_1(R) = 1/2$ ,  $e_2(R) = 1/3$ , and  $e_3(R) = 3/4$  (and correspondingly  $e_1(B) = 1/2$ ,  $e_2(B) = 2/3$ , and  $e_3(B) = 1/4$ ), state transition matrix  $p_{ij} = \begin{pmatrix} .3 & .6 & .1 \\ .5 & .2 & .3 \\ .4 & .1 & .5 \end{pmatrix}$ , and initial state probabilities  $w_i = 1/3$ . Suppose we observe the sequence  $RBR$ , then we can find the “optimal” state sequence to explain this sequence of observations by running the Viterbi algorithm by hand:



In the first step, we initialize the probabilities at  $t = 1$  to  $v_j(t = 1) = w_j e_j(R)$  for each  $j = 1, 2, 3$ . These are given in the first column to the left, as  $1/6, 1/9, 1/4$ , respectively.

In the second step,  $t = 2$ , we determine first  $v_1(t = 2)$  by considering the three quantities  $v_i(1)p_{i1}$  for  $i = 1, 2, 3$ . They are respectively  $(1/6) \cdot .3, (1/9) \cdot .5$ , and  $(1/4) \cdot .4$ . The third one is the largest, so according to the algorithm we set  $v_1(2) = [(1/4) \cdot .4] \cdot (1/2) = .05$ , and remember that the maximum probability path to state  $j = 1$  at time  $t = 2$  came from state  $j = 3$  at time  $t = 1$  (blue line). Similarly, to determine  $v_2(2)$  we consider the three quantities  $v_i(1)p_{i2}$  for  $i = 1, 2, 3$ , respectively  $(1/6) \cdot .6, (1/9) \cdot .2, (1/4) \cdot .1$ , and the first is the largest, so we set  $v_2(2) = [(1/6) \cdot .6] \cdot (2/3) = .06$ . Finally, to determine  $v_3(2)$  we consider the three quantities  $v_i(1)p_{i3}$  for  $i = 1, 2, 3$ , respectively  $(1/6) \cdot .1, (1/9) \cdot .3, (1/4) \cdot .5$ , and the third is the largest, so we set  $v_3(2) = [(1/4) \cdot .5] \cdot (1/4) = .03125$ .

In the third step,  $t = 3$ , we determine first  $v_1(t = 3)$  by considering the three quantities  $v_i(2)p_{i1}$  for  $i = 1, 2, 3$ . They are respectively  $.05 \cdot .3, .06 \cdot .5$ , and  $.03125 \cdot .4$ . The second is the largest, so according to the algorithm we set  $v_1(3) = [.06 \cdot .5] \cdot (1/2) = .01\bar{6}$ , and remember that the maximum probability path to state  $j = 1$  at time  $t = 3$  came from state  $j = 2$  at time  $t = 2$  (blue line). Similarly, to determine  $v_2(3)$  we consider the three quantities  $v_i(2)p_{i2}$  for  $i = 1, 2, 3$ , respectively  $.05 \cdot .6, .06 \cdot .2, .03125 \cdot .1$ , and the first is the largest, so we set  $v_2(3) = [.05 \cdot .6] \cdot (1/3) = .01$ . Finally, to determine  $v_3(3)$  we consider the three quantities  $v_i(2)p_{i3}$  for  $i = 1, 2, 3$ , respectively  $.05 \cdot .1, .06 \cdot .3, .03125 \cdot .5$ , and the second is the largest, so we set  $v_3(3) = [.06 \cdot .3] \cdot (3/4) = .015$ .

Since there are only three observations, we can now use the termination step to determine that the maximum probability for the observations  $O = RBR$  is  $P^* = .01\bar{6}$  with state path  $Q = 1, 2, 1$  (purple lines).

## 8.5 Conditional Random Fields (CRFs)

unknown words

While the HMM is a useful and powerful model, it turns out that HMMs need a number of augmentations to achieve high accuracy. For example, in POS tagging as in other tasks, we often run into **unknown words**: proper names and acronyms are created very often, and even new common nouns and verbs enter the language at a surprising rate. It would be great to have ways to add arbitrary features to help with this, perhaps based on capitalization or morphology (words starting with capital letters are likely to be proper nouns, words ending with *-ed* tend to be past tense (VBD or VBN), etc.) Or knowing the previous or following words might be a useful feature (if the previous word is *the*, the current tag is unlikely to be a verb).

Although we could try to hack the HMM to find ways to incorporate some of these, in general it's hard for generative models like HMMs to add arbitrary features directly into the model in a clean way. We've already seen a model for combining arbitrary features in a principled way: log-linear models like the logistic regression model of Chapter 5! But logistic regression isn't a sequence model; it assigns a class to a single observation.

Luckily, there is a discriminative sequence model based on log-linear models: the **conditional random field (CRF)**. We'll describe here the **linear chain CRF**, the version of the CRF most commonly used for language processing, and the one whose conditioning closely matches the HMM.

Assuming we have a sequence of input words  $X = x_1 \dots x_n$  and want to compute a sequence of output tags  $Y = y_1 \dots y_n$ . In an HMM to compute the best tag sequence that maximizes  $P(Y|X)$  we rely on Bayes' rule and the likelihood  $P(X|Y)$ :

$$\begin{aligned}\hat{Y} &= \operatorname{argmax}_Y p(Y|X) \\ &= \operatorname{argmax}_Y p(X|Y)p(Y) \\ &= \operatorname{argmax}_Y \prod_i p(x_i|y_i) \prod_i p(y_i|y_{i-1})\end{aligned}\tag{8.21}$$

In a CRF, by contrast, we compute the posterior  $p(Y|X)$  directly, training the CRF

to discriminate among the possible tag sequences:

$$\hat{Y} = \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} P(Y|X) \quad (8.22)$$

However, the CRF does not compute a probability for each tag at each time step. Instead, at each time step the CRF computes log-linear functions over a set of relevant features, and these local features are aggregated and normalized to produce a global probability for the whole sequence.

Let's introduce the CRF more formally, again using  $X$  and  $Y$  as the input and output sequences. A CRF is a log-linear model that assigns a probability to an entire output (tag) sequence  $Y$ , out of all possible sequences  $\mathcal{Y}$ , given the entire input (word) sequence  $X$ . We can think of a CRF as like a giant version of what multinomial logistic regression does for a single token. Recall that the feature function  $f$  in regular multinomial logistic regression can be viewed as a function of a tuple: a token  $x$  and a label  $y$  (page 89). In a CRF, the function  $F$  maps an entire input sequence  $X$  and an entire output sequence  $Y$  to a feature vector. Let's assume we have  $K$  features, with a weight  $w_k$  for each feature  $F_k$ :

$$p(Y|X) = \frac{\exp\left(\sum_{k=1}^K w_k F_k(X, Y)\right)}{\sum_{Y' \in \mathcal{Y}} \exp\left(\sum_{k=1}^K w_k F_k(X, Y')\right)} \quad (8.23)$$

It's common to also describe the same equation by pulling out the denominator into a function  $Z(X)$ :

$$p(Y|X) = \frac{1}{Z(X)} \exp\left(\sum_{k=1}^K w_k F_k(X, Y)\right) \quad (8.24)$$

$$Z(X) = \sum_{Y' \in \mathcal{Y}} \exp\left(\sum_{k=1}^K w_k F_k(X, Y')\right) \quad (8.25)$$

We'll call these  $K$  functions  $F_k(X, Y)$  **global features**, since each one is a property of the entire input sequence  $X$  and output sequence  $Y$ . We compute them by decomposing into a sum of **local** features for each position  $i$  in  $Y$ :

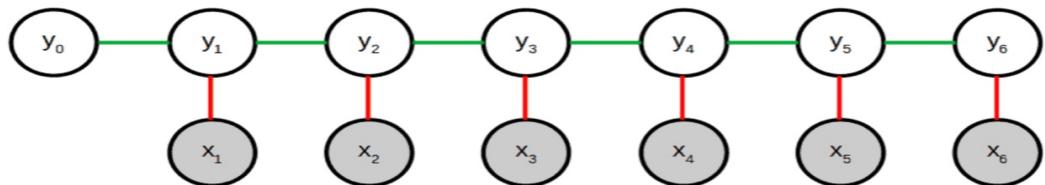
$$F_k(X, Y) = \sum_{i=1}^n f_k(y_{i-1}, y_i, X, i) \quad (8.26)$$

Each of these local features  $f_k$  in a linear-chain CRF is allowed to make use of the current output token  $y_i$ , the previous output token  $y_{i-1}$ , the entire input string  $X$  (or any subpart of it), and the current position  $i$ . This constraint to only depend on the current and previous output tokens  $y_i$  and  $y_{i-1}$  are what characterizes a **linear chain CRF**. As we will see, this limitation makes it possible to use versions of the efficient Viterbi and Forward-Backwards algorithms from the HMM. A general CRF, by contrast, allows a feature to make use of any output token, and are thus necessary for tasks in which the decision depend on distant output tokens, like  $y_{i-4}$ . General CRFs require more complex inference, and are less commonly used for language processing.

### 3.1. Feature Functions

CRFs are undirected graphical models, for which we define features (i.e., feature functions) manually. Simply put, **feature functions are descriptions of words depending on their position in the sequence and their surrounding words**. For example, "The word is a question mark and the first word of the sequence is a verb.", "This word is a noun and the previous word is a noun", or "This is a pronoun and the next word is a verb". After selecting the feature functions, we find their weights by training the CRF.

Here's a schematic representation of a CRF:



**There are two types of feature functions: the state and transition functions.** State functions (red in the above graph) represent the connection between hidden variables and the observed ones. Transition functions (green) do the same for the hidden states themselves. We can define an arbitrary number of feature functions depending on the specific task we want to perform.

For instance, for our previous example of POS tagging, one feature can be whether a word and its predecessor are of the same type. Another one could be whether the current word is the end of the sequence.

### 8.5.3 Inference and Training for CRFs

How do we find the best tag sequence  $\hat{Y}$  for a given input  $X$ ? We start with Eq. 8.22:

$$\begin{aligned} \hat{Y} &= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} P(Y|X) \\ &= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \frac{1}{Z(X)} \exp \left( \sum_{k=1}^K w_k F_k(X, Y) \right) \end{aligned} \quad (8.27)$$

$$= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \exp \left( \sum_{k=1}^K w_k \sum_{i=1}^n f_k(y_{i-1}, y_i, X, i) \right) \quad (8.28)$$

$$= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \sum_{k=1}^K w_k \sum_{i=1}^n f_k(y_{i-1}, y_i, X, i) \quad (8.29)$$

$$= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \sum_{i=1}^n \sum_{k=1}^K w_k f_k(y_{i-1}, y_i, X, i) \quad (8.30)$$

We can ignore the exp function and the denominator  $Z(X)$ , as we do above, because exp doesn't change the argmax, and the denominator  $Z(X)$  is constant for a given observation sequence  $X$ .

How should we decode to find this optimal tag sequence  $\hat{y}$ ? Just as with HMMs, we'll turn to the Viterbi algorithm, which works because, like the HMM, the linear-chain CRF depends at each timestep on only one previous output token  $y_{i-1}$ .

Concretely, this involves filling an  $N \times T$  array with the appropriate values, maintaining backpointers as we proceed. As with HMM Viterbi, when the table is filled, we simply follow pointers back from the maximum value in the final column to retrieve the desired set of labels.

The requisite changes from HMM Viterbi have to do only with how we fill each cell. Recall from Eq. 8.19 that the recursive step of the Viterbi equation computes the Viterbi value of time  $t$  for state  $j$  as

$$v_t(j) = \max_{i=1}^N v_{t-1}(i) a_{ij} b_j(o_t); \quad 1 \leq j \leq N, 1 < t \leq T \quad (8.31)$$

which is the HMM implementation of

$$v_t(j) = \max_{i=1}^N v_{t-1}(i) P(s_j|s_i) P(o_t|s_j) \quad 1 \leq j \leq N, 1 < t \leq T \quad (8.32)$$

The CRF requires only a slight change to this latter formula, replacing the  $a$  and  $b$  prior and likelihood probabilities with the CRF features:

$$v_t(j) = \max_{i=1}^N v_{t-1}(i) + \sum_{k=1}^K w_k f_k(y_{t-1}, y_t, X, t) \quad 1 \leq j \leq N, 1 < t \leq T \quad (8.33)$$

Learning in CRFs relies on the same supervised learning algorithms we presented for logistic regression. Given a sequence of observations, feature functions, and corresponding outputs, we use stochastic gradient descent to train the weights to maximize the log-likelihood of the training corpus. The local nature of linear-chain CRFs means that the forward-backward algorithm introduced for HMMs in Appendix A can be extended to a CRF version that will efficiently compute the necessary derivatives. As with logistic regression, L1 or L2 regularization is important.

## 3.2. Math

In mathematical terms, let's say that  $f_k(y_t, y_{t-1}, \mathbf{x}_t)$  are our feature functions, with  $y_t$ ,  $y_{t-1}$ , and  $\mathbf{x}_t$  being current label, previous label and current word, respectively. **For each word or element of a sequence, a feature can be 0 or 1, depending on whether the word has the corresponding feature.** Then, CRFs assign weights  $\theta_k$  to the feature functions based on the training data. As a result, we get a score of  $\sum_{k=1}^K \theta_k f_k(y_t, y_{t-1}, \mathbf{x}_t)$  for each word. To make this a probability distribution, we exponentiate and normalize it over all the possible feature functions. Then, multiplying the probabilities of each word, we get the final formula:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{t=1}^T \exp \left\{ \sum_{k=1}^K \theta_k f_k(y_t, y_{t-1}, \mathbf{x}_t) \right\}$$

The normalization factor  $Z(\mathbf{x})$  accounts for all the possible feature functions. It might be very difficult to compute as the number of these functions can explode. As a result, we resort to **approximate estimation** of  $Z(\mathbf{x})$ .

## 3.3. Example

Let's consider the same part-of-speech (POS) tagging problem from above. For a CRF model, we need to define some feature functions first. Then, using our training examples, we estimate the weight of each function for each word. We can come up with various functions. Here, we'll consider the following ones:

1.  $f_1$ : the word ends with *ing* and it's a noun
2.  $f_2$ : the word is the second word and it's a verb
3.  $f_3$ : the previous word is a verb and the current word is a noun
4.  $f_4$ : the word belongs to  $\{I, you, he, she, we, they\}$  and it's a pronoun
5.  $f_5$ : the word is in  $\{my, your, his, her, their, our\}$  and it's an adjective
6.  $f_6$ : the word comes after a possessive adjective and it's a noun
7.  $f_7$ : the word comes after the verb *is* and it's an adjective

Using the training data, we learn the weights of all the features for all the words. In practice, we use **Gradient Descent** to find the weights that maximize the sequence's probability, i.e., minimize its log-likelihood.

## 4. HMMs vs. CRFs

We can use both models for the same tasks such as POS tagging. However, their approaches differ.

## 4. HMMs vs. CRFs

We can use both models for the same tasks such as [POS tagging](#). However, their approaches differ.

**HMMs first learn the joint distribution of the observed and hidden variables during training.** Then, to do prediction, they use the Bayes rule to compute the conditional probability. In contrast, CRFs directly learn conditional probability.

The training objective of HMMs is [Maximum Likelihood Estimation \(MLE\)](#) by counting. Therefore, they aim to maximize the probability of the observed data. Conversely, we train CRFs using a gradient-based method, which gives us a hyperplane that classifies new data.

For inference, both models use the same algorithm called the [forward-backward algorithm](#).

**HMMs are more constrained than CRFs** since, in HMMs, each state depends on a fixed set of previous hidden states. This enables them to capture the local context. In contrast, CRFs can consider non-consecutive states as a feature function. That's why they can capture both local and global contexts.

For example, the feature function for the first word in the sentence might inspect the relationship between the start and the end of a sequence, labeling the first word as a verb if the sequence ends with a question mark. That isn't possible with HMMs. As a result, CRFs can be more accurate than HMMs if we define the right feature functions.

**We represent HMMs with directed graphs.** They are a special case of Bayesian networks which possess the [Markov property](#). However, **we represent CRFs with undirected graphs.** As a result, they do not hold the Markov property. In other words,  $y_t$  in a CRF doesn't make  $y_{t+1}$  independent from  $y_{t-1}, y_{t-2}, \dots, y_1$ .

With that said, **CRFs are computationally more challenging.** The reason is that the normalization factor is intractable and its exact computation is impossible or hard to do.