Natural Language Processing

TF-IDF:

TF: term frequency

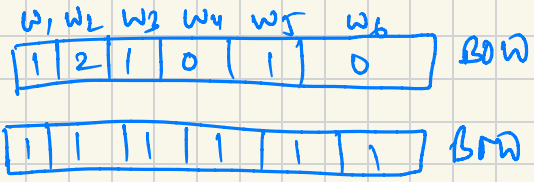
IDF: Inverse Document frequency.

r_1: $w_1\ w_2\ w_3\ w_2\ w_5$ — ⑤

r_2 $w_1\ w_3\ w_4\ w_5\ w_6\ w_2$ — ⑥

:

r_N



$$TF(w_2, r_1) = 2/5$$

$$TF(w_i, r_j) = \frac{\#\text{ of times } w_i \text{ occurs in } r_j}{\text{Total \# of words in } r_j}$$

So, $TF(w_i, r_j)$ will be range of 1.

$$\boxed{0 \le TF(w_i, r_j) \le 1}$$

IDF:

$$IDF(w_i, D_c) = \log\left(\frac{N}{n_i}\right)$$

N → # docs

n_i → # docs which contain w_i.

Here, $n_i \le N \Rightarrow \frac{N}{n_i} \ge 1$

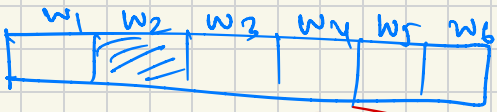
if $n_i \uparrow \Rightarrow \frac{N}{n_i} \downarrow \Rightarrow \log\left(\frac{N}{n_i}\right) \downarrow$

→ Monotonic function

if w_i is more frequent → IDF will be low

if w_j is rare in doc → IDF will be higher.

Now Compute TF-IDF:



$TF(w_2, r_1) * IDF(w_2, D_c)$

if w_j is frequent in r_i

$TF(w_j, r_i)$ will be higher.

if w_j is rare in D_c

$IDF(w_j, D_c)$ will be higher.

Here, we are giving more weightage to w_j if it is rare in the corpus or more frequent in a review.

Problem: still it can't solve semantic meaning problem.

Question

In a corpus of 10000 documents you randomly pick a document, say D, which has a total of 250 words and the word 'data' occurs 20 times. Also, the word 'data' occurs in 2500 (out of 10000) documents. What will be the tfidf entry for the term 'data' in a bag of words vector representation for D.

$TF = \frac{20}{250}$ $\quad IDF = \ln\left(\frac{10000}{2500}\right) = \ln(4)$

$TF\text{-}IDF(data) = \frac{2}{25} \times 1.3863$

$= .11$

You have the following three documents - D1, D2, D3:
D1: Natural language processing is becoming important since soon we will begin talking to our computers.
D2: If computers understand natural language they will become much simpler to use.
D3: Speech recognition is the first step to build computers like us.
Answer the following with respect to the above set of 3 documents after text normalization (stop word removal and lemmatization) has been done on all 3 documents.

What is the vocabulary V?

V =(become, build, computer, first, important, language, like, natural, processing, recognition, simple, speech, step, talk, understand, us, use)

What are the number of bigrams and trigrams in D2?

D2 after normalization looks as follows:
computer understand natural language become simple use

Bi-grams and trigrams are extracted by sliding windows of size 2 and 3 respectively over the sentence. So, if n is the length of the sentence then No. of bigrams=(n – 1) where n ≥ 2 and No. of trigrams=(n – 2) when n ≥ 3. So, we get 6 bigrams and 5 trigrams. Of course, this answer will differ if your normalized document has more or less words.

What will be the BoW document vector for document D3 if we are using a tf based document vector?

Document D3 after normalization:
speech recognition first step build computer like us
Bag-of-words vector with normalized tf:
81(0,1,1,1,0,0,1,0,0,1,0,1,1,0,0,1,0)
All words occur only once in D3. So, if you use plain tf then the factor 18 will not be present.

Suppose you have the following two 4-dimensional word vectors for two words w1 and w2 respec- tively:
w1 = (0.2, 0.1, 0.3, 0.4) and w2 = (0.3, 0, 0.2, 0.5)
What is the cosine similarity between w1 and w2? Are the words w1 and w2 similar or dissimilar?

$$\cos(\theta) = \frac{\langle w_1, w_2 \rangle}{\|w_1\| \, \|w_2\|}$$

$$\Rightarrow \frac{6 + 0 + 6 + 20}{\sqrt{4 + 1 + 9 + 16}\ \sqrt{9 + 0 + 4 + 25}}$$

$$\Rightarrow \frac{32}{\sqrt{30}\ \sqrt{38}} \approx \frac{16}{17} \approx 1 \quad (\text{close to } 1)$$

So, they are very similar.

1) sequence labeling.

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 xi in an input word sequence, a label yi, 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).

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

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 x1,x2,...,xn of (tokenized) words and a tagset, and the output is a sequence y1,y2,...,yn of tags, each output yi corresponding exactly to one input xi, 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).

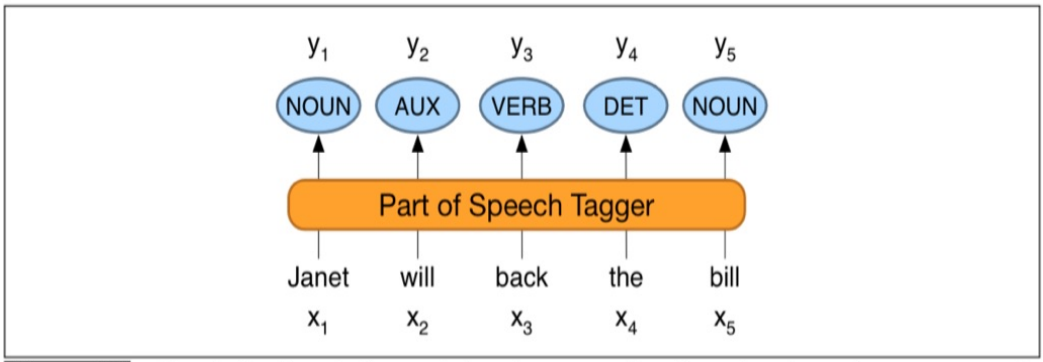


Figure 8.3 The task of part-of-speech tagging: mapping from input words $x_1, x_2, \ldots, x_n$ to output POS tags $y_1, y_2, \ldots, y_n$.

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).

$$\hat{y} = \underset{y}{\operatorname{argmax}}\, P(y|x)$$

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

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 $\underset{y}{\operatorname{argmax}}\, P(y|x)$ 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

[1] 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.

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.

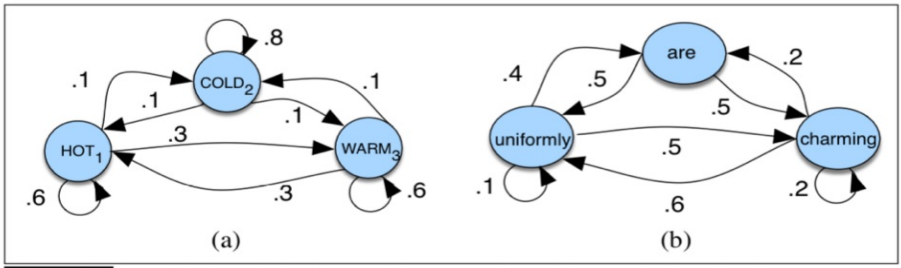


Figure 8.8 A Markov chain for weather (a) and one for words (b), showing states and transitions. A start distribution π is required; setting $\pi = [0.1, 0.7, 0.2]$ for (a) would mean a probability 0.7 of starting in state 2 (cold), probability 0.1 of starting in state 1 (hot), etc.

Markov assumption

More formally, consider a sequence of state variables $q_1, q_2, ..., q_i$. 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.

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

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 $w_1 ... w_t$. 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:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} a_{12} \ldots a_{N1} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$
$\pi = \pi_1, \pi_2, \ldots, \pi_N$	an **initial probability distribution** over states. π_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.

hidden

len Markov model

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:

$Q = q_1 q_2 \ldots q_N$	a set of N **states**
$A = a_{11} \ldots a_{ij} \ldots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$
$O = o_1 o_2 \ldots o_T$	a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, \ldots, v_V$
$B = b_i(o_t)$	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
$\pi = \pi_1, \pi_2, \ldots, \pi_N$	an **initial probability distribution** over states. π_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:

Markov Assumption: $P(q_i|q_1, \ldots, q_{i-1}) = P(q_i|q_{i-1})$ (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:

Output Independence: $P(o_i|q_1, \ldots q_i, \ldots, q_T, o_1, \ldots, o_i, \ldots, o_T) = P(o_i|q_i)$ (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:

transition probability

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})} \tag{8.8}$$

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

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

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

emission probability

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)} \tag{8.10}$$

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

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

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 $P(\text{MD}|\text{will})$. Instead, $P(\text{will}|\text{MD})$ 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

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

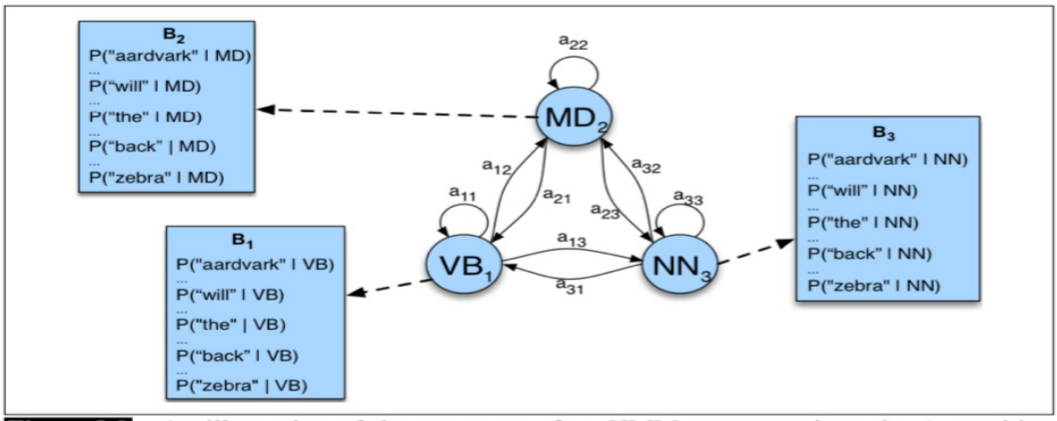


Figure 8.9 An illustration of the two parts of an HMM representation: the A transition probabilities used to compute the prior probability, and the B observation likelihoods that are associated with each state, one likelihood for each possible observation word.

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 $w_1 \dots w_n$:

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

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

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

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

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

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:

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

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;

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

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:

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

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^n → no of tokens in an input sentence.
→ no of POS tags

∝

It is too huge.

↓

viterbi algorithm is efficient to calculate.

It has computational cost : $O(n^2 * n)$ (Dynamic programming).

The Viterbi algorithm

function VITERBI(*observations* of len T,*state-graph* of len N) **returns** *best-path*, *path-prob*

create a path probability matrix *viterbi*[N,T]
for each state s **from** 1 **to** N **do**
    $viterbi[s,1] \leftarrow \pi_s * b_s(o_1)$ — initialization
    $backpointer[s,1] \leftarrow 0$ — initialization
for each time step t **from** 2 **to** T **do** — recursion
  for each state s **from** 1 **to** N **do** — recursion
    $viterbi[s,t] \leftarrow \max_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$ ← $v_t(j) = \max_{i=1}^{N} v_{t-1}(i) a_{ij} b_j(o_t)$
    $backpointer[s,t] \leftarrow \operatorname{argmax}_{s'=1}^{N} viterbi[s',t-1] * a_{s',s} * b_s(o_t)$
$bestpathprob \leftarrow \max_{s=1}^{N} viterbi[s,T]$ — termination
$bestpathpointer \leftarrow \operatorname{argmax}_{s=1}^{N} viterbi[s,T]$ — termination
bestpath ← the path starting at state *bestpathpointer*, that follows backpointer[] to states back in time
return *bestpath*, *bestpathprob*

time Complexity = $O(n^2 t)$
↓
for each time { state to state calculation }

Hidden Markov Models and the Viterbi algorithm

An HMM $H = \big(p_{ij}, e_i(a), w_i\big)$ is understood to have N hidden Markov states labelled by i $(1 \le i \le N)$, and M possible observables for each state, labelled by a $(1 \le a \le M)$. The state transition probabilies are $p_{ij} = p(q_{t+1} = j \,|\, q_t = i)$, $1 \le i, j \le N$ (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 $H = \big(p_{ij}, e_i(a), w_i\big)$, 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.,

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

and let w_i be the initial probabilities of the states i at time $t = 1$. (Note this notation avoids the frightening greek letters δ, π, and λ 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

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

transition probability from state i to state j

previous viterbi path probability

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

together with initialization

$$v_i(1) = w_i\, e_i(O_1) \qquad 1 \le i \le N$$

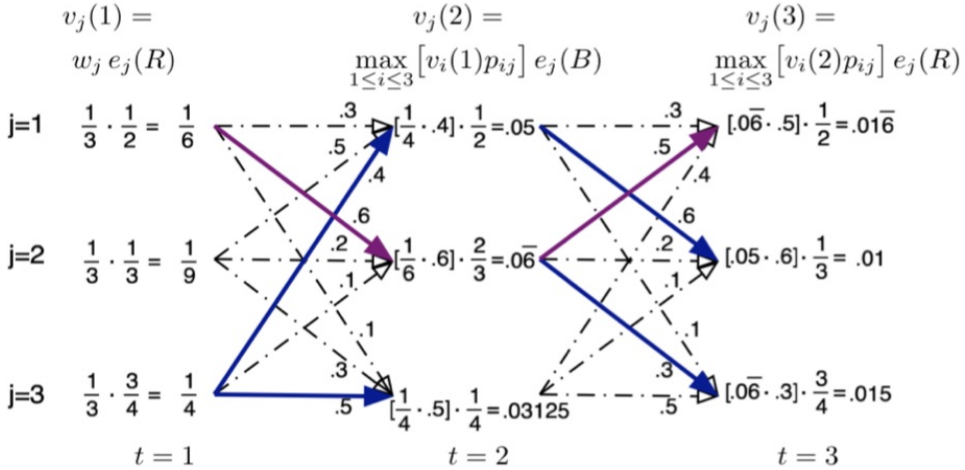
and termination

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

(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 $v_2(2) = [(1/6)\cdot.6]\cdot(2/3) = .0\bar{6}$. 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 $.05\cdot.3$, $.0\bar{6}\cdot.5$, and $.03125\cdot.4$. The second is the largest, so according to the algorithm we set $v_1(3) = [.0\bar{6}\cdot.5]\cdot(1/2) = .01\bar{6}$, 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 $.05\cdot.6$, $.0\bar{6}\cdot.2$, $.03125\cdot.1$, and the first is the largest, so we set $v_2(3) = [.05\cdot.6]\cdot(1/3) = .01$. Finally, to determine $v_3(2)$ we consider the three quantities $v_i(1)p_{i3}$ for $i = 1, 2, 3$, respectively $.05\cdot.1$, $.0\bar{6}\cdot.3$, $.03125\cdot.5$, and the second is the largest, so we set $v_3(3) = [.0\bar{6}\cdot.3]\cdot(3/4) = .015$.

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).

For the following question parts, assume that the log probabilities (log base 2) are as follows (these are rounded so they don't quite normalize as you would expect):

Initial	
N	−1
V	−1

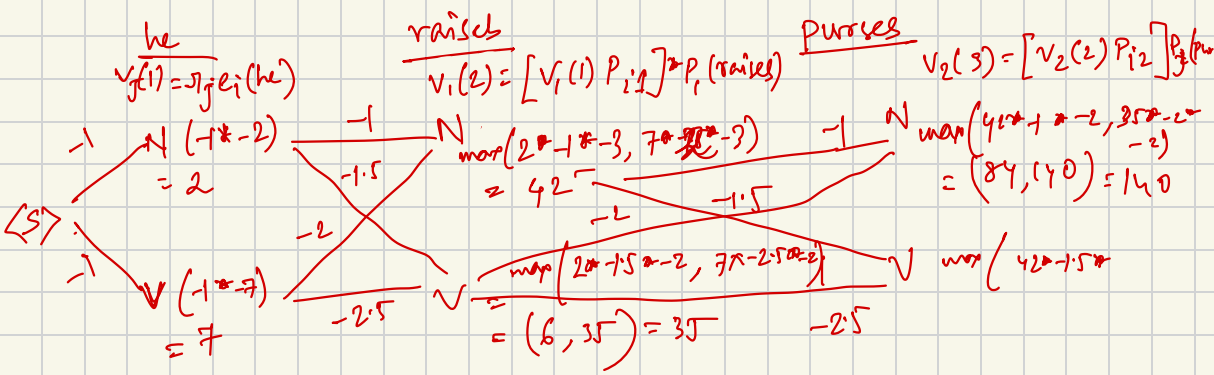
Transitions			
	N	V	STOP
y_{i-1} = N	−1	−1.5	−2
y_{i-1} = V	−2	−2.5	−0.5

Emissions					
	their	he	my	raises	purses
N	−4	−2	−3	−3	−2
V	−5	−7	−5	−2	−4

Consider the sentence *he raises purses*

d. (6 points) Draw the Viterbi chart for this data. Be sure to fill in every value. **You do not need to include a final column for the STOP symbol; just fill in the chart up through the third word.**

N: -3, -7, -10
V: -8, -6.5, -12.5



e. (4 points) What is the highest posterior probability tag sequence for this sentence (now including the STOP transition)? **Is this sequence consistent with your interpretation of the sentence above?**

NNN with log probability -12, which is not consistent with our interpretation

f. (2 points) What is the minimum beam size needed to get the same answer as Viterbi on this particular example? Justify your answer in no more than one sentence.

2, since just strictly maxing gives you NVN

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.

CRF

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 $X = x_1 ... x_n$ and want to compute a sequence of output tags $Y = y_1 ... y_n$. 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:

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

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:

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

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

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

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

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:

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

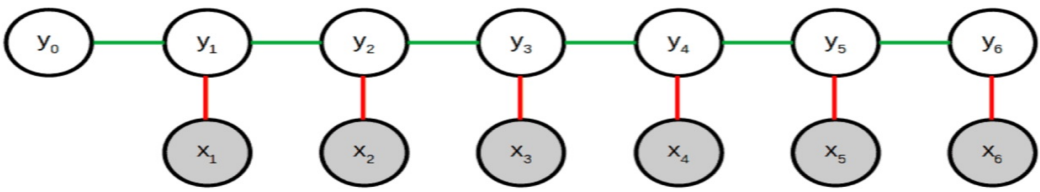
Each of these local features f_k in a linear-chain CRF is allowed to make use of the current output token y_i, the previous output token y_{i-1}, the entire input string X (or any subpart of it), and the current position i. This constraint to only depend on the current and previous output tokens y_i 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

linear chain CRF

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:

$$\begin{aligned}
\hat{Y} &= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} P(Y|X) \\
&= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \frac{1}{Z(X)} \exp\left(\sum_{k=1}^{K} w_k F_k(X,Y)\right) && (8.27) \\
&= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \exp\left(\sum_{k=1}^{K} w_k \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i)\right) && (8.28) \\
&= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \sum_{k=1}^{K} w_k \sum_{i=1}^{n} f_k(y_{i-1}, y_i, X, i) && (8.29) \\
&= \underset{Y \in \mathcal{Y}}{\operatorname{argmax}} \sum_{i=1}^{n} \sum_{k=1}^{K} w_k f_k(y_{i-1}, y_i, X, i) && (8.30)
\end{aligned}$$

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

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

which is the HMM implementation of

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

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

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

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.

Suppose each tag has at most k < t legal transitions it can make.
What is the runtime of
this modified version of Viterbi?

$O(nKT)$

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 θ_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:

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

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.

HMM - generative model
CRF → discriminative model.

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 $y_{t-1}, y_{t-2}, \cdots, y_1$.

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.

Q1

Version A: I, V (8) Compare using an HMM for sequence labeling to using logistic regression to independently predict each tag. Which of the following is an advantage of HMMs compared to logistic regression (so only choose options which apply to HMMs and not LR). Circle all that apply:

I. They allow you to incorporate structural information about adjacent tag pairs.

II. They allow you to use "non-local" features on the input (e.g., the current tag can depend directly on words that are far away in the sentence).

III. They support inference that is asymptotically faster in the number of possible tags.

IV. They are discriminative rather than generative and so directly model P (y|x), which is what we care most about.

V. They are generative rather than discriminative so we can draw samples of sentences from the distri- bution P (y, x).

Q2

Version A: I (9) Compare using a CRF for sequence labeling to using logistic regression to independently predict each tag. Which of the following is an advantage of CRFs compared to logis- tic regression (only choose options which apply to CRFs and not LR). Circle all that apply:

I. They allow you to incorporate structural information about adjacent tag pairs.

II. They allow you to use "non-local" features on the input (e.g., the current tag can depend directly on words that are far away in the sentence).

III. They support inference that is asymptotically faster in the number of possible tags.

IV. They are discriminative rather than generative and so directly model P (y|x), which is what we care most about.

V. They are generative rather than discriminative so we can draw samples of sentences from the distri- bution P (y, x).

Q3

Consider the following corpus:

their/N raises/N

he/N raises/V

my/N purses/N

a. (2 points) List the maximum-likelihood initial state probabilities for an HMM estimated from this data.

Here, two states → {N, V}

we can see N can be a starting state but V can't be, so

$\sum_{i=1}^{n} \pi_i = 1$ so, $P(N) = 1$

$P(V) = 0$.

Q4

List the maximum-likelihood transition probabilities for an HMM estimated from this data (including transitions to the STOP symbol).

	N	V	STOP
N	3/5	1/5	2/5
V	0	0	1

Singular value Decomposition:

$$A = U \Sigma V^T$$

$\Rightarrow$ The columns of U are orthonormal eigenvectors of AA^T

$\Rightarrow$ The columns of V are orthogonal eigenvectors of A^TA

$\Rightarrow$ Eigenvalues $\lambda_1, \lambda_2 \cdots \lambda_r$ of AA^T are the eigenvalues of A^TA.

$$\sigma_i = \sqrt{\lambda_i}$$

$$\Sigma^i = \text{diag}(\sigma_1, \sigma_2 \cdots \sigma_n)$$

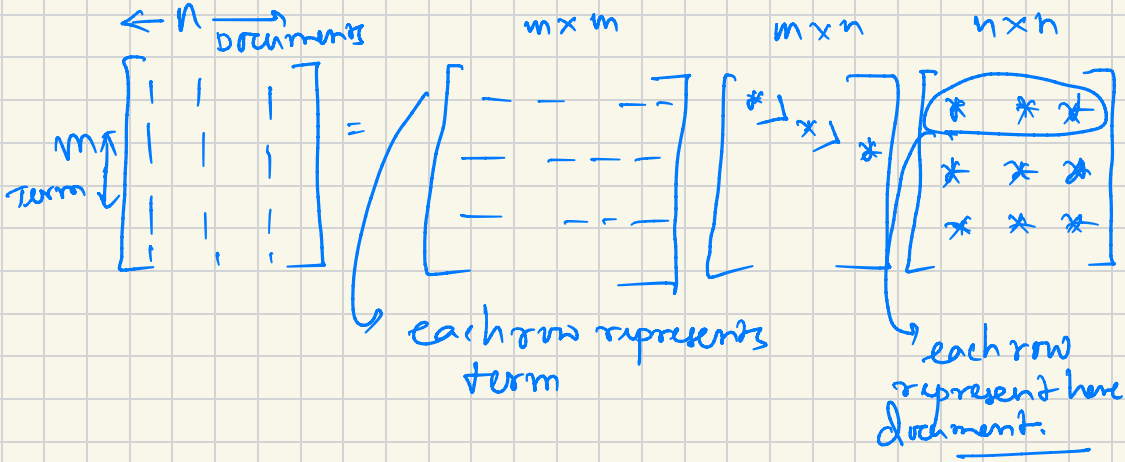
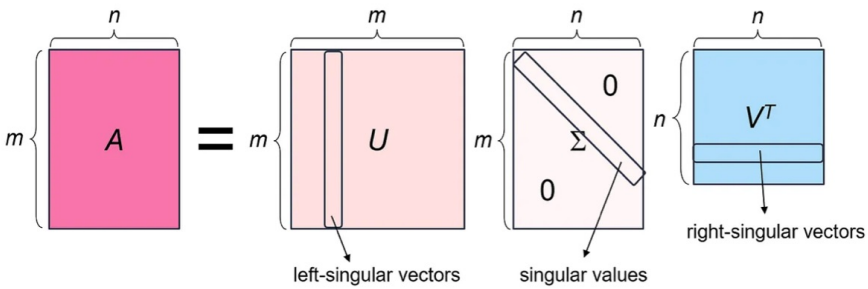


Image by Peggy und Marco Lachmann-Anke from Pixabay

Mathematical Definition

The singular value decomposition of an $m \times n$ real matrix A is a factorization of the form $A = U\Sigma V^t$, where:

- U is an $m \times m$ **orthogonal matrix** (i.e., its columns and rows are orthonormal vectors). The columns of U are called the **left-singular vectors** of A.
- Σ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal. The diagonal entries $\sigma_i = \Sigma_{ii}$ are known as the **singular values** of A and are typically arranged in descending order, i.e., $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$. The number of the non-zero singular values is equal to the rank of A.
- V is an $n \times n$ orthogonal matrix. The columns of V are called the **right-singular vectors** of A.



SVD of a matrix

Every matrix has a singular value decomposition (a proof of this statement can be found here). This is unlike eigenvalue decomposition, for example, which can be applied only to squared diagonalizable matrices.