Tagging with Hidden Markov Models

Michael Collins

1 Tagging Problems

In many NLP problems, we would like to model *pairs* of sequences. Part-of-speech (POS) tagging is perhaps the earliest, and most famous, example of this type of problem. In POS tagging our goal is to build a model whose input is a *sentence*, for example

the dog saw a cat

and whose output is a *tag sequence*, for example

D N V D N (1)

(here we use D for a determiner, N for noun, and V for verb). The tag sequence is the same length as the input sentence, and therefore specifies a single tag for each word in the sentence (in this example D for *the*, N for *dog*, V for *saw*, and so on).

We will use $x_1 \dots x_n$ to denote the input to the tagging model: we will often refer to this as a *sentence*. In the above example we have the length $n = 5$, and $x_1 =$ *the*, $x_2 =$ *dog*, $x_3 =$ *saw*, $x_4 =$ *the*, $x_5 =$ *cat*. We will use $y_1 \dots y_n$ to denote the output of the tagging model: we will often refer to this as the *state sequence* or *tag sequence*. In the above example we have $y_1 =$ D, $y_2 =$ N, $y_3 =$ V, and so on.

This type of problem, where the task is to map a sentence $x_1 \dots x_n$ to a tag sequence $y_1 \dots y_n$, is often referred to as a **sequence labeling problem**, or a **tagging problem**.

We will assume that we have a set of *training examples*, $(x^{(i)}, y^{(i)})$ for $i = 1 \dots m$, where each $x^{(i)}$ is a sentence $x_1^{(i)} \dots x_{n_i}^{(i)}$, and each $y^{(i)}$ is a tag sequence $y_1^{(i)} \dots y_{n_i}^{(i)}$ (we assume that the i'th example is of length n_i). Hence $x_j^{(i)}$ is the j'th word in the i'th training example, and $y_j^{(i)}$ is the tag for that word. Our task is to learn a function that maps sentences to tag sequences from these training examples.

2 Generative Models, and The Noisy Channel Model

Supervised problems in machine learning are defined as follows. We assume training examples $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$, where each example consists of an input $x^{(i)}$ paired with a label $y^{(i)}$. We use $\mathcal{X}$ to refer to the set of possible inputs, and $\mathcal{Y}$ to refer to the set of possible labels. Our task is to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that maps any input x to a label $f(x)$.

Many problems in natural language processing are supervised learning problems. For example, in tagging problems each $x^{(i)}$ would be a sequence of words $x_1^{(i)} \dots x_{n_i}^{(i)}$, and each $y^{(i)}$ would be a sequence of tags $y_1^{(i)} \dots y_{n_i}^{(i)}$ (we use n_i to refer to the length of the i'th training example). $\mathcal{X}$ would refer to the set of all sequences $x_1 \dots x_n$, and $\mathcal{Y}$ would be the set of all tag sequences $y_1 \dots y_n$. Our task would be to learn a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that maps sentences to tag sequences. In machine translation, each input x would be a sentence in the source language (e.g., Chinese), and each "label" would be a sentence in the target language (e.g., English). In speech recognition each input would be the recording of some utterance (perhaps pre-processed using a Fourier transform, for example), and each label is an entire sentence. Our task in all of these examples is to learn a function from inputs x to labels y, using our training examples $(x^{(i)}, y^{(i)})$ for $i = 1 \dots n$ as evidence.

One way to define the function $f(x)$ is through a *conditional model*. In this approach we define a model that defines the conditional probability

$$p(y|x)$$

for any x, y pair. The parameters of the model are estimated from the training examples. Given a new test example x, the output from the model is

$$f(x) = \arg\max_{y \in \mathcal{Y}} p(y|x)$$

Thus we simply take the most likely label y as the output from the model. If our model $p(y|x)$ is close to the true conditional distribution of labels given inputs, the function $f(x)$ will be close to optimal.

An alternative approach, which is often used in machine learning and natural language processing, is to define a *generative model*. Rather than directly estimating the conditional distribution $p(y|x)$, in generative models we instead model the *joint* probability

$$p(x, y)$$

over (x, y) pairs. The parameters of the model $p(x, y)$ are again estimated from the training examples $(x^{(i)}, y^{(i)})$ for $i = 1 \dots n$. In many cases we further decompose

the probability $p(x, y)$ as follows:

$$p(x, y) = p(y)p(x|y) \tag{2}$$

and then estimate the models for $p(y)$ and $p(x|y)$ separately. These two model components have the following interpretations:

- $p(y)$ is a *prior* probability distribution over labels y.
- $p(x|y)$ is the probability of generating the input x, given that the underlying label is y.

We will see that in many cases it is very convenient to decompose models in this way; for example, the classical approach to speech recognition is based on this type of decomposition.

Given a generative model, we can use Bayes rule to derive the conditional probability $p(y|x)$ for any (x, y) pair:

$$p(y|x) = \frac{p(y)p(x|y)}{p(x)}$$

where

$$p(x) = \sum_{y \in \mathcal{Y}} p(x, y) = \sum_{y \in \mathcal{Y}} p(y)p(x|y)$$

Thus the joint model is quite versatile, in that we can also derive the probabilities $p(x)$ and $p(y|x)$.

We use Bayes rule directly in applying the joint model to a new test example. Given an input x, the output of our model, $f(x)$, can be derived as follows:

$$\begin{aligned} f(x) &= \arg\max_y p(y|x) \\ &= \arg\max_y \frac{p(y)p(x|y)}{p(x)} \qquad (3) \\ &= \arg\max_y p(y)p(x|y) \qquad (4) \end{aligned}$$

Eq. 3 follows by Bayes rule. Eq. 4 follows because the denominator, $p(x)$, does not depend on y, and hence does not affect the $\arg\max$. This is convenient, because it means that we do not need to calculate $p(x)$, which can be an expensive operation.

Models that decompose a joint probability into into terms $p(y)$ and $p(x|y)$ are often called *noisy-channel* models. Intuitively, when we see a test example x, we assume that has been generated in two steps: first, a label y has been chosen with probability $p(y)$; second, the example x has been generated from the distribution

$p(x|y)$. The model $p(x|y)$ can be interpreted as a "channel" which takes a label y as its input, and corrupts it to produce x as its output. Our task is to find the most likely label y, given that we observe x.

In summary:

- Our task is to learn a function from inputs x to labels $y = f(x)$. We assume training examples $(x^{(i)}, y^{(i)})$ for $i = 1 \ldots n$.
- In the noisy channel approach, we use the training examples to estimate models $p(y)$ and $p(x|y)$. These models define a joint (generative) model

$$p(x, y) = p(y)p(x|y)$$

- Given a new test example x, we predict the label

$$f(x) = \arg\max_{y \in \mathcal{Y}} p(y)p(x|y)$$

  Finding the output $f(x)$ for an input x is often referred to as the *decoding* problem.

3 Generative Tagging Models

We now see how generative models can be applied to the tagging problem. We assume that we have a finite vocabulary $\mathcal{V}$, for example $\mathcal{V}$ might be the set of words seen in English, e.g., $\mathcal{V} = \{$*the, dog, saw, cat, laughs*$, \ldots\}$. We use $\mathcal{K}$ to denote the set of possible tags; again, we assume that this set is finite. We then give the following definition:

Definition 1 (Generative Tagging Models) *Assume a finite set of words $\mathcal{V}$, and a finite set of tags $\mathcal{K}$. Define $\mathcal{S}$ to be the set of all sequence/tag-sequence pairs $\langle x_1 \ldots x_n, y_1 \ldots y_n \rangle$ such that $n \geq 0$, $x_i \in \mathcal{V}$ for $i = 1 \ldots n$ and $y_i \in \mathcal{K}$ for $i = 1 \ldots n$. A* generative tagging model *is then a function p such that:*

1. *For any $\langle x_1 \ldots x_n, y_1 \ldots y_n \rangle \in \mathcal{S}$,*

$$p(x_1 \ldots x_n, y_1 \ldots y_n) \geq 0$$

2. *In addition,*

$$\sum_{\langle x_1 \ldots x_n, y_1 \ldots y_n \rangle \in \mathcal{S}} p(x_1 \ldots x_n, y_1 \ldots y_n) = 1$$

Hence $p(x_1 \ldots x_n, y_1 \ldots y_n)$ is a probability distribution over pairs of sequences (i.e., a probability distribution over the set $\mathcal{S}$).

Given a generative tagging model, the function from sentences $x_1 \ldots x_n$ to tag sequences $y_1 \ldots y_n$ is defined as

$$f(x_1 \ldots x_n) = \arg\max_{y_1 \ldots y_n} p(x_1 \ldots x_n, y_1 \ldots y_n)$$

Thus for any input $x_1 \ldots x_n$, we take the highest probability tag sequence as the output from the model. □

Having introduced generative tagging models, there are three critical questions:

- How we define a generative tagging model $p(x_1 \ldots x_n, y_1 \ldots y_n)$?
- How do we estimate the parameters of the model from training examples?
- How do we efficiently find

$$\arg\max_{y_1 \ldots y_n} p(x_1 \ldots x_n, y_1 \ldots y_n)$$

  for any input $x_1 \ldots x_n$?

The next section describes how trigram hidden Markov models can be used to answer these three questions.

4 Trigram Hidden Markov Models (Trigram HMMs)

In this section we describe an important type of generative tagging model, a *trigram hidden Markov model*, describe how the parameters of the model can be estimated from training examples, and describe how the most likely sequence of tags can be found for any sentence.

4.1 Definition of Trigram HMMs

We now give a formal definition of trigram hidden Markov models (trigram HMMs). The next section shows how this model form is derived, and gives some intuition behind the model.

Definition 2 (Trigram Hidden Markov Model (Trigram HMM)) *A trigram HMM consists of a finite set $\mathcal{V}$ of possible words, and a finite set $\mathcal{K}$ of possible tags, together with the following parameters:*

- *A parameter*

$$q(s|u,v)$$

for any trigram (u,v,s) *such that* $s \in \mathcal{K} \cup \{\textit{STOP}\}$*, and* $u,v \in \mathcal{V} \cup \{*\}$*. The value for* $q(s|u,v)$ *can be interpreted as the probability of seeing the tag* s *immediately after the bigram of tags* (u,v)*.*

- *A parameter*

$$e(x|s)$$

for any $x \in \mathcal{V}$*,* $s \in \mathcal{K}$*. The value for* $e(x|s)$ *can be interpreted as the probability of seeing observation* x *paired with state* s*.*

Define $\mathcal{S}$ *to be the set of all sequence/tag-sequence pairs* $\langle x_1 \ldots x_n, y_1 \ldots y_{n+1} \rangle$ *such that* $n \geq 0$*,* $x_i \in \mathcal{V}$ *for* $i = 1 \ldots n$*,* $y_i \in \mathcal{K}$ *for* $i = 1 \ldots n$*, and* $y_{n+1} = \textit{STOP}$*.*

We then define the probability for any $\langle x_1 \ldots x_n, y_1 \ldots y_{n+1} \rangle \in \mathcal{S}$ *as*

$$p(x_1 \ldots x_n, y_1 \ldots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i|y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i|y_i)$$

where we have assumed that $y_0 = y_{-1} = *$*.* $\square$

As one example, if we have $n = 3$, $x_1 \ldots x_3$ equal to the sentence *the dog laughs*, and $y_1 \ldots y_4$ equal to the tag sequence `D N V STOP`, then

$$\begin{aligned} p(x_1 \ldots x_n, y_1 \ldots y_{n+1}) \quad = \quad & q(\texttt{D}|*,*) \times q(\texttt{N}|*,\texttt{D}) \times q(\texttt{V}|\texttt{D},\texttt{N}) \times q(\texttt{STOP}|\texttt{N},\texttt{V}) \\ & \times e(\textit{the}|\texttt{D}) \times e(\textit{dog}|\texttt{N}) \times e(\textit{laughs}|\texttt{V}) \end{aligned}$$

Note that this model form is a noisy-channel model. The quantity

$$q(\texttt{D}|*,*) \times q(\texttt{N}|*,\texttt{D}) \times q(\texttt{V}|\texttt{D},\texttt{N}) \times q(\texttt{STOP}|\texttt{N},\texttt{V})$$

is the prior probability of seeing the tag sequence `D N V STOP`, where we have used a second-order Markov model (a trigram model), very similar to the language models we derived in the previous lecture. The quantity

$$e(\textit{the}|\texttt{D}) \times e(\textit{dog}|\texttt{N}) \times e(\textit{laughs}|\texttt{V})$$

can be interpreted as the conditional probability $p(\textit{the dog laughs}|\texttt{D N V STOP})$: that is, the conditional probability $p(x|y)$ where x is the sentence *the dog laughs*, and y is the tag sequence `D N V STOP`.

4.2 Independence Assumptions in Trigram HMMs

We now describe how the form for trigram HMMs can be derived: in particular, we describe the independence assumptions that are made in the model. Consider a pair of sequences of random variables $X_1 \dots X_n$, and $Y_1 \dots Y_n$, where n is the length of the sequences. We assume that each X_i can take any value in a finite set $\mathcal{V}$ of *words*. For example, $\mathcal{V}$ might be a set of possible words in English, for example $\mathcal{V} = \{$*the, dog, saw, cat, laughs*$, \dots\}$. Each Y_i can take any value in a finite set $\mathcal{K}$ of possible *tags*. For example, $\mathcal{K}$ might be the set of possible part-of-speech tags for English, e.g. $\mathcal{K} = \{\texttt{D}, \texttt{N}, \texttt{V}, \dots\}$.

The length n is itself a random variable—it can vary across different sentences—but we will use a similar technique to the method used for modeling variable-length Markov processes (see the previous lecture notes).

Our task will be to model the joint probability

$$P(X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_n = y_n)$$

for any observation sequence $x_1 \dots x_n$ paired with a state sequence $y_1 \dots y_n$, where each x_i is a member of $\mathcal{V}$, and each y_i is a member of $\mathcal{K}$.

We will find it convenient to define one additional random variable Y_{n+1}, which always takes the value STOP. This will play a similar role to the STOP symbol seen for variable-length Markov sequences, as described in the previous lecture notes.

The key idea in hidden Markov models is the following definition:

$$\begin{aligned} & P(X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) \\ = & \prod_{i=1}^{n+1} P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) \prod_{i=1}^{n} P(X_i = x_i | Y_i = y_i) \end{aligned} \quad (5)$$

where we have assumed that $y_0 = y_{-1} =$ *, where * is a special start symbol.

Note the similarity to our definition of trigram HMMs. In trigram HMMs we have made the assumption that the joint probability factorizes as in Eq. 5, and in addition we have assumed that for any i, for any values of y_{i-2}, y_{i-1}, y_i,

$$P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1}) = q(y_i | y_{i-2}, y_{i-1})$$

and that for any value of i, for any values of x_i and y_i,

$$P(X_i = x_i | Y_i = y_i) = e(x_i | y_i)$$

Eq. 5 can be derived as follows. First, we can write

$$\begin{aligned} & P(X_1 = x_1 \dots X_n = x_n, Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) \\ = & P(Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) P(X_1 = x_1 \dots X_n = x_n | Y_1 = y_1 \dots Y_{n+1} = y_{n+1}) \end{aligned} \quad (6)$$

This step is exact, by the chain rule of probabilities. Thus we have decomposed the joint probability into two terms: first, the probability of choosing tag sequence $y_1 \ldots y_{n+1}$; second, the probability of choosing the word sequence $x_1 \ldots x_n$, conditioned on the choice of tag sequence. Note that this is exactly the same type of decomposition as seen in noisy channel models.

Now consider the probability of seeing the tag sequence $y_1 \ldots y_{n+1}$. We make independence assumptions as follows: we assume that for any sequence $y_1 \ldots y_{n+1}$,

$$P(Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) = \prod_{i=1}^{n+1} P(Y_i = y_i | Y_{i-2} = y_{i-2}, Y_{i-1} = y_{i-1})$$

That is, we have assumed that the sequence $Y_1 \ldots Y_{n+1}$ is a second-order Markov sequence, where each state depends only on the previous two states in the sequence.

Next, consider the probability of the word sequence $x_1 \ldots x_n$, conditioned on the choice of tag sequence, $y_1 \ldots y_{n+1}$. We make the following assumption:

$$\begin{aligned} & P(X_1 = x_1 \ldots X_n = x_n | Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) \\ = & \prod_{i=1}^{n} P(X_i = x_i | X_1 = x_1 \ldots X_{i-1} = x_{i-1}, Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) \\ = & \prod_{i=1}^{n} P(X_i = x_i | Y_i = y_i) \end{aligned} \tag{7}$$

The first step of this derivation is exact, by the chain rule. The second step involves an independence assumption, namely that for $i = 1 \ldots n$,

$$P(X_i = x_i | X_1 = x_1 \ldots X_{i-1} = x_{i-1}, Y_1 = y_1 \ldots Y_{n+1} = y_{n+1}) = P(X_i = x_i | Y_i = y_i)$$

Hence we have assumed that the value for the random variable X_i depends only on the value of Y_i. More formally, the value for X_i is conditionally independent of the previous observations $X_1 \ldots X_{i-1}$, and the other state values $Y_1 \ldots Y_{i-1}, Y_{i+1} \ldots Y_{n+1}$, given the value of Y_i.

One useful way of thinking of this model is to consider the following stochastic process, which generates sequence pairs $y_1 \ldots y_{n+1}, x_1 \ldots x_n$:

1. Initialize $i = 1$ and $y_0 = y_{-1} =$ *.
2. Generate y_i from the distribution
$$q(y_i | y_{i-2}, y_{i-1})$$
3. If $y_i =$ STOP then return $y_1 \ldots y_i, x_1 \ldots x_{i-1}$. Otherwise, generate x_i from the distribution
$$e(x_i | y_i),$$
set $i = i + 1$, and return to step 2.

4.3 Estimating the Parameters of a Trigram HMM

We will assume that we have access to some training data. The training data consists of a set of examples where each example is a sentence $x_1 \dots x_n$ paired with a tag sequence $y_1 \dots y_n$. Given this data, how do we estimate the parameters of the model? We will see that there is a simple and very intuitive answer to this question.

Define $c(u, v, s)$ to be the number of times the sequence of three states (u, v, s) is seen in training data: for example, $c(\text{V, D, N})$ would be the number of times the sequence of three tags V, D, N is seen in the training corpus. Similarly, define $c(u, v)$ to be the number of times the tag bigram (u, v) is seen. Define $c(s)$ to be the number of times that the state s is seen in the corpus. Finally, define $c(s \rightsquigarrow x)$ to be the number of times state s is seen paired sith observation x in the corpus: for example, $c(\text{N} \rightsquigarrow \text{dog})$ would be the number of times the word *dog* is seen paired with the tag N.

Given these definitions, the *maximum-likelihood* estimates are

$$q(s|u, v) = \frac{c(u, v, s)}{c(u, v)}$$

and

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)}$$

For example, we would have the estimates

$$q(\texttt{N}|\texttt{V}, \texttt{D}) = \frac{c(\texttt{V}, \texttt{D}, \texttt{N})}{c(\texttt{V}, \texttt{D})}$$

and

$$e(\textit{dog}|\texttt{N}) = \frac{c(\texttt{N} \rightsquigarrow \textit{dog})}{c(\texttt{N})}$$

Thus estimating the parameters of the model is simple: we just read off counts from the training corpus, and then compute the maximum-likelihood estimates as described above.

4.4 Decoding with HMMs: the Viterbi Algorithm

We now turn to the problem of finding the most likely tag sequence for an input sentence $x_1 \dots x_n$. This is the problem of finding

$$\arg \max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

where the $\arg\max$ is taken over all sequences $y_1 \dots y_{n+1}$ such that $y_i \in \mathcal{K}$ for $i = 1 \dots n$, and $y_{n+1} = \text{STOP}$. We assume that p again takes the form

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i | y_i) \tag{8}$$

Recall that we have assumed in this definition that $y_0 = y_{-1} = *$, and $y_{n+1} = \text{STOP}$.

The naive, brute force method would be to simply enumerate all possible tag sequences $y_1 \dots y_{n+1}$, score them under the function p, and take the highest scoring sequence. For example, given the input sentence

the dog barks

and assuming that the set of possible tags is $\mathcal{K} = \{\texttt{D}, \texttt{N}, \texttt{V}\}$, we would consider all possible tag sequences:

D	D	D	STOP
D	D	N	STOP
D	D	V	STOP
D	N	D	STOP
D	N	N	STOP
D	N	V	STOP
...			

and so on. There are $3^3 = 27$ possible sequences in this case.

For longer sentences, however, this method will be hopelessly inefficient. For an input sentence of length n, there are $|\mathcal{K}|^n$ possible tag sequences. The exponential growth with respect to the length n means that for any reasonable length sentence, brute-force search will not be tractable.

4.4.1 The Basic Algorithm

Instead, we will see that we can efficiently find the highest probability tag sequence, using a dynamic programming algorithm that is often called *the Viterbi algorithm*. The input to the algorithm is a sentence $x_1 \dots x_n$. Given this sentence, for any $k \in \{1 \dots n\}$, for any sequence $y_1 \dots y_k$ such that $y_i \in \mathcal{K}$ for $i = 1 \dots k$ we define the function

$$r(y_1 \dots y_k) = \prod_{i=1}^{k} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{k} e(x_i | y_i) \tag{9}$$

This is simply a truncated version of the definition of p in Eq. 8, where we just consider the first k terms. In particular, note that

$$\begin{aligned} p(x_1 \dots x_n, y_1 \dots y_{n+1}) &= r(y_1 \dots y_n) \times q(y_{n+1}|y_{n-1}, y_n) \\ &= r(y_1 \dots y_n) \times q(\text{STOP}|y_{n-1}, y_n) \end{aligned} \quad (10)$$

Next, for any any $k \in \{1 \dots n\}$, for any $u \in \mathcal{K}$, $v \in \mathcal{K}$, define $S(k, u, v)$ to be the set of sequences $y_1 \dots y_k$ such that $y_{k-1} = u$, $y_k = v$, and $y_i \in \mathcal{K}$ for $i = 1 \dots k$. Thus $S(k, u, v)$ is the set of all tag sequences of length k, which end in the tag bigram (u, v). Define

$$\pi(k, u, v) = \max_{\langle y_1 \dots y_k \rangle \in S(k,u,v)} r(y_1 \dots y_k) \quad (11)$$

We now observe that we can calculate the $\pi(k, u, v)$ values for all (k, u, v) efficiently, as follows. First, as a base case define

$$\pi(0, *, *) = 1$$

and

$$\pi(0, u, v) = 0$$

if $u \neq *$ or $v \neq *$. These definitions just reflect the fact that we always assume that $y_0 = y_{-1} = *$.

Next, we give the recursive definition.

Proposition 1 *For any $k \in \{1 \dots n\}$, for any $u \in \mathcal{K}$ and $v \in \mathcal{K}$, we can use the following recursive definition:*

$$\pi(k, u, v) = \max_{w \in \mathcal{K}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v)) \quad (12)$$

This definition is recursive because the definition makes use of the $\pi(k-1, w, v)$ values computed for shorter sequences. This definition will be key to our dynamic programming algorithm.

How can we justify this recurrence? Recall that $\pi(k, u, v)$ is the highest probability for any sequence $y_1 \dots y_k$ ending in the bigram (u, v). Any such sequence must have $y_{k-2} = w$ for some state w. The highest probability for any sequence of length $k-1$ ending in the bigram (w, u) is $\pi(k-1, w, u)$, hence the highest probability for any sequence of length k ending in the trigram (w, u, v) must be

$$\pi(k-1, w, u) \times q(v|w, u) \times e(x_i|v)$$

In Eq. 12 we simply search over all possible values for w, and return the maximum.

Our second claim is the following:

Input: a sentence $x_1 \dots x_n$, parameters $q(s|u,v)$ and $e(x|s)$.
Initialization: Set $\pi(0,*,*) = 1$, and $\pi(0,u,v) = 0$ for all (u,v) such that $u \neq *$ or $v \neq *$.
Algorithm:

- For $k = 1 \dots n$,
  - For $u \in \mathcal{K}$, $v \in \mathcal{K}$,

$$\pi(k,u,v) = \max_{w \in \mathcal{K}} (\pi(k-1,w,u) \times q(v|w,u) \times e(x_k|v))$$

- **Return** $\max_{u \in \mathcal{K}, v \in \mathcal{K}} (\pi(n,u,v) \times q(\text{STOP}|u,v))$

Figure 1: The basic Viterbi Algorithm.

Proposition 2

$$\max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \max_{u \in \mathcal{K}, v \in \mathcal{K}} (\pi(n,u,v) \times q(\textit{STOP}|u,v)) \quad (13)$$

This follows directly from the identity in Eq. 10.

Figure 1 shows an algorithm that puts these ideas together. The algorithm takes a sentence $x_1 \dots x_n$ as input, and returns

$$\max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

as its output. The algorithm first fills in the $\pi(k,u,v)$ values in using the recursive definition. It then uses the identity in Eq. 13 to calculate the highest probability for any sequence.

The running time for the algorithm is $O(n|\mathcal{K}|^3)$, hence it is linear in the length of the sequence, and cubic in the number of tags.

4.4.2 The Viterbi Algorithm with Backpointers

The algorithm we have just described takes a sentence $x_1 \dots x_n$ as input, and returns

$$\max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

as its output. However we'd really like an algorithm that returned the highest probability sequence, that is, an algorithm that returns

$$\arg \max_{y_1 \dots y_{n+1}} p(x_1 \dots x_n, y_1 \dots y_{n+1})$$

Input: a sentence $x_1 \ldots x_n$, parameters $q(s|u,v)$ and $e(x|s)$.
Initialization: Set $\pi(0,*,*) = 1$, and $\pi(0,u,v) = 0$ for all (u,v) such that $u \neq *$ or $v \neq *$.
Algorithm:

- For $k = 1 \ldots n$,
  - For $u \in \mathcal{K}$, $v \in \mathcal{K}$,

$$\begin{aligned} \pi(k,u,v) &= \max_{w \in \mathcal{K}} (\pi(k-1,w,u) \times q(v|w,u) \times e(x_k|v)) \\ bp(k,u,v) &= \arg\max_{w \in \mathcal{K}} (\pi(k-1,w,u) \times q(v|w,u) \times e(x_k|v)) \end{aligned}$$

- Set $(y_{n-1}, y_n) = \arg\max_{(u,v)} (\pi(n,u,v) \times q(\text{STOP}|u,v))$
- For $k = (n-2) \ldots 1$,

$$y_k = bp(k+2, y_{k+1}, y_{k+2})$$

- **Return** the tag sequence $y_1 \ldots y_n$

Figure 2: The Viterbi Algorithm with backpointers.

for any input sentence $x_1 \ldots x_n$.

Figure 2 shows a modified algorithm that achieves this goal. The key step is to store backpointer values $bp(k,u,v)$ at each step, which record the previous state w which leads to the highest scoring sequence ending in (u,v) at position k (the use of backpointers such as these is very common in dynamic programming methods). At the end of the algorithm, we unravel the backpointers to find the highest probability sequence, and then return this sequence. The algorithm again runs in $O(n|\mathcal{K}|^3)$ time.

5 Summary

We've covered a fair amount of material in this note, but the end result is fairly straightforward: we have derived a complete method for learning a tagger from a training corpus, and for applying it to new sentences. The main points were as follows:

- A trigram HMM has parameters $q(s|u,v)$ and $e(x|s)$, and defines the joint

probability of any sentence $x_1 \dots x_n$ paired with a tag sequence $y_1 \dots y_{n+1}$ (where $y_{n+1} = \text{STOP}$) as

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i | y_i)$$

- Given a training corpus from which we can derive counts, the maximum-likelihood estimates for the parameters are

$$q(s|u,v) = \frac{c(u,v,s)}{c(u,v)}$$

and

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)}$$

- Given a new sentence $x_1 \dots x_n$, and parameters q and e that we have estimated from a training corpus, we can find the highest probability tag sequence for $x_1 \dots x_n$ using the algorithm in figure 2 (the Viterbi algorithm).

Convergence Proof for the Perceptron Algorithm

Michael Collins

Figure 1 shows the perceptron learning algorithm, as described in lecture. In this note we give a convergence proof for the algorithm (also covered in lecture).

The convergence theorem is as follows:

Theorem 1 *Assume that there exists some parameter vector* $\underline{\theta}^*$ *such that* $||\underline{\theta}^*|| = 1$*, and some* $\gamma > 0$ *such that for all* $t = 1 \ldots n$*,*

$$y_t(\underline{x}_t \cdot \underline{\theta}^*) \geq \gamma$$

Assume in addition that for all $t = 1 \ldots n$*,* $||\underline{x}_t|| \leq R$*.*

Then the perceptron algorithm makes at most

$$\frac{R^2}{\gamma^2}$$

errors. (The definition of an error is as follows: an error occurs whenever we have $y' \neq y_t$ *for some* (j, t) *pair in the algorithm.)*

Note that for any vector $\underline{x}$, we use $||\underline{x}||$ to refer to the Euclidean norm of $\underline{x}$, i.e., $||\underline{x}|| = \sqrt{\sum_i x_i^2}$.

Proof: First, define $\underline{\theta}^k$ to be the parameter vector when the algorithm makes its k'th error. Note that we have

$$\underline{\theta}^1 = \underline{0}$$

Next, assuming the k'th error is made on example t, we have

$$\begin{aligned} \underline{\theta}^{k+1} \cdot \underline{\theta}^* &= (\underline{\theta}^k + y_t \underline{x}_t) \cdot \underline{\theta}^* && (1) \\ &= \underline{\theta}^k \cdot \underline{\theta}^* + y_t \underline{x}_t \cdot \underline{\theta}^* && (2) \\ &\geq \underline{\theta}^k \cdot \underline{\theta}^* + \gamma && (3) \end{aligned}$$

Eq. 1 follows by the definition of the perceptron updates. Eq. 3 follows because by the assumptions of the theorem, we have

$$y_t \underline{x}_t \cdot \underline{\theta}^* \geq \gamma$$

Definition: $\text{sign}(z) = 1$ if $z \geq 0$, -1 otherwise.

Inputs: number of iterations, T; training examples $(\underline{x}_t, y_t)$ for $t \in \{1 \dots n\}$ where $\underline{x} \in \mathbb{R}^d$ is an input, and $y_t \in \{-1, +1\}$ is a label.

Initialization: $\underline{\theta} = \underline{0}$ (i.e., all parameters are set to 0)

Algorithm:

- For $j = 1 \dots T$
  - For $t = 1 \dots n$
    1. $y' = \text{sign}(\underline{x}_t \cdot \underline{\theta})$
    2. If $y' \neq y_t$ Then $\underline{\theta} = \underline{\theta} + y_t \underline{x}_t$, Else leave $\underline{\theta}$ unchanged

Output: parameters $\underline{\theta}$

Figure 1: The perceptron learning algorithm.

It follows by induction on k (recall that $||\underline{\theta}^1|| = 0$), that

$$\underline{\theta}^{k+1} \cdot \underline{\theta}^* \geq k\gamma$$

In addition, because $||\underline{\theta}^{k+1}|| \times ||\underline{\theta}^*|| \geq \underline{\theta}^{k+1} \cdot \underline{\theta}^*$, and $||\underline{\theta}^*|| = 1$, we have

$$||\underline{\theta}^{k+1}|| \geq k\gamma \tag{4}$$

In the second part of the proof, we will derive an upper bound on $||\underline{\theta}^{k+1}||$. We have

$$\begin{aligned} ||\underline{\theta}^{k+1}||^2 &= ||\underline{\theta}^k + y_t \underline{x}_t||^2 && (5) \\ &= ||\underline{\theta}^k||^2 + y_t^2 ||\underline{x}_t||^2 + 2 y_t \underline{x}_t \cdot \underline{\theta}^k && (6) \\ &\leq ||\underline{\theta}^k||^2 + R^2 && (7) \end{aligned}$$

The equality in Eq. 5 follows by the definition of the perceptron updates. Eq. 7 follows because we have: 1) $y_t^2 ||\underline{x}_t||^2 = ||\underline{x}_t||^2 \leq R^2$ by the assumptions of the theorem, and because $y_t^2 = 1$; 2) $y_t \underline{x}_t \cdot \underline{\theta}^k \leq 0$ because we know that the parameter vector $\underline{\theta}^k$ gave an error on the t^{th} example.

It follows by induction on k (recall that $||\underline{\theta}^1||^2 = 0$), that

$$||\underline{\theta}^{k+1}||^2 \leq kR^2 \tag{8}$$

Combining the bounds in Eqs. 4 and 8 gives

$$k^2\gamma^2 \leq ||\underline{\theta}^{k+1}||^2 \leq kR^2$$

from which it follows that

$$k \leq \frac{R^2}{\gamma^2}$$

□

Log-Linear Models, MEMMs, and CRFs

Michael Collins

1 Notation

Throughout this note I'll use *underline* to denote vectors. For example, $\underline{w} \in \mathbb{R}^d$ will be a vector with components $w_1, w_2, \ldots w_d$. We use $\exp(x)$ for the exponential function, i.e., $\exp(x) = e^x$.

2 Log-linear models

We have sets $\mathcal{X}$ and $\mathcal{Y}$: we will assume that $\mathcal{Y}$ is a finite set. Our goal is to build a model that estimates the conditional probability $p(y|x)$ of a label $y \in \mathcal{Y}$ given an input $x \in \mathcal{X}$. For example, x might be a word, and y might be a candidate part-of-speech (noun, verb, preposition etc.) for that word. We have a feature-vector definition $\underline{\phi} : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^d$. We also assume a parameter vector $\underline{w} \in \mathbb{R}^d$. Given these definitions, log-linear models take the following form:

$$p(y|x; \underline{w}) = \frac{\exp\left(\underline{w} \cdot \underline{\phi}(x, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\underline{w} \cdot \underline{\phi}(x, y')\right)}$$

This is the conditional probability of y given x, under parameters $\underline{w}$.

Some motivation for this expression is as follows. The inner product

$$\underline{w} \cdot \underline{\phi}(x, y)$$

can take any value (positive or negative), and can be interpreted as being a measure of the plausibility of label y given input x. For a given input x, we can calculate this inner product for each possible label $y \in \mathcal{Y}$. We'd like to transform these quantities into a well-formed distribution $p(y|x)$. If we exponentiate the inner product,

$$\exp\left(\underline{w} \cdot \underline{\phi}(x, y)\right)$$

we have a strictly positive quantity—i.e., a value that is greater than 0. Finally, by dividing by the normalization constant

$$\sum_{y' \in \mathcal{Y}} \exp\left(\underline{w} \cdot \underline{\phi}(x, y')\right)$$

we ensure that $\sum_{y \in \mathcal{Y}} p(y|x; w) = 1$. Hence we have gone from inner products $\underline{w} \cdot \underline{\phi}(x, y)$, which can take either positive or negative values, to a probability distribution.

An important question is how the parameters $\underline{w}$ can be estimated from data. We turn to this question next.

The Log-Likelihood Function. To estimate the parameters, we assume that we have a set of n labeled examples, $\{(x_i, y_i)\}_{i=1}^n$. The *log-likelihood function* is

$$L(\underline{w}) = \sum_{i=1}^{n} \log p(y_i|x_i; \underline{w})$$

We can think of $L(\underline{w})$ as being a function that for a given $\underline{w}$ measures how well $\underline{w}$ explains the labeled examples. A "good" value for $\underline{w}$ will give a high value for $p(y_i|x_i; \underline{w})$ for all $i = 1 \ldots n$, and thus will have a high value for $L(\underline{w})$.

The *maximum-likelihood estimates* are

$$\underline{w}^* = \arg\max_{\underline{w} \in \mathbb{R}^d} \sum_{i=1}^{n} \log p(y_i|x_i; \underline{w})$$

The maximum-likelihood estimates are thus the parameters that best fit the training set, under the criterion $L(\underline{w})$.[1]

Finding the maximum-likelihood estimates. So given a training set $\{(x_i, y_i)\}_{i=1}^n$, how do we find the maximum-likelihood parameter estimates $\underline{w}^*$? Unfortunately, an analytical solution does not in general exist. Instead, people generally use gradient-based methods to optimize $L(w)$. The simplest method, "vanilla" gradient ascent, takes roughly the following form:

1. Set $\underline{w}^0$ to some initial value, for example set $w_j^0 = 0$ for $j = 1 \ldots d$
2. For $t = 1 \ldots T$:

[1] In some cases this maximum will not be well-defined—intuitively, some parameter values may diverge to $+\infty$ or $-\infty$—but for now we'll assume that the maximum exists, and that all parameters take finite values at the maximum.

- For $j = 1 \ldots d$, set

$$w_j^t = w_j^{t-1} + \alpha_t \times \frac{\partial}{\partial w_j} L(\underline{w}^{t-1})$$

where $\alpha_t > 0$ is some stepsize, and $\frac{\partial}{\partial w_j} L(\underline{w}^{t-1})$ is the derivative of L with respect to w_j.

3. Return the final parameters $\underline{w}^T$.

Thus at each iteration we calculate the gradient at the current point $\underline{w}^{t-1}$, and move some distance in the direction of the gradient.

In practice, more sophisticated optimization methods are used: one common to choice is to use L-BFGS, a quasi-newton method. We won't go into the details of these optimization methods in the course: the good news is that good software packages are available for methods such as L-BFGS. Implementations of L-BFGS will generally require us to calculate the value of the objective function $L(\underline{w})$, and the value of the partial derivatives, $\frac{\partial}{\partial w_j} L(\underline{w})$, at any point $\underline{w}$. Fortunately, this will be easy to do.

So what form do the partial derivatives take? A little bit of calculus gives

$$\frac{\partial}{\partial w_j} L(\underline{w}) = \sum_i \phi_j(x_i, y_i) - \sum_i \sum_y p(y|x_i; \underline{w}) \phi_j(x_i, y)$$

The first sum in the expression, $\sum_i \phi_j(x_i, y_i)$, is the sum of the j'th feature value $\phi_j(x_i, y_i)$ across the labeled examples $\{(x_i, y_i)\}_{i=1}^n$. The second sum again involves a sum over the training examples, but for each training example we calculate the *expected* feature value, $\sum_y p(y|x_i; \underline{w}) \phi_j(x_i, y)$. Note that this expectation is taken with respect to the distribution $p(y|x_i; \underline{w})$ under the current parameter values $\underline{w}$.

Regularized log-likelihood. In many applications, it has been shown to be highly beneficial to modify the log-likelihood function to include an additional *regularization term*. The modified criterion is then

$$L(\underline{w}) = \sum_{i=1}^n \log p(y_i|x_i; \underline{w}) - \frac{\lambda}{2} ||\underline{w}||^2$$

where $||\underline{w}||^2 = \sum_j w_j^2$, and $\lambda > 0$ is parameter dictating the strength of the regularization term. We will again choose our parameter values to be

$$\underline{w}^* = \arg\max_{\underline{w} \in \mathbb{R}^d} L(\underline{w})$$

Note that we now have a trade-off when estimating the parameters: we will try to make the $\log p(y_i|x_i;\underline{w})$ terms as high as possible, but at the same time we'll try to keep the norm $||\underline{w}||^2$ small (the larger the value of λ, the smaller we will require the norm to be). The regularization term penalizes large parameter values.

Intuitively, we can think of the $||\underline{w}||^2$ term as being a penalty on "complexity" of the model, where the larger the parameters are, the more complex the model is. We'd like to find a model that fits the data well, but that also has low complexity.[2]

In practice, the regularization term has been found to be very useful in building log-linear models, in particular in cases where the number of parameters, d, is large. This scenario is very common in natural language processing applications. It is not uncommon for the number of parameters d to be far larger than the number of training examples n, and in this case we can often still achieve good generalization performance, as long as a regularizer is used to penalize large values of $||\underline{w}||^2$. (There are close connections to support vector machines, where linear models are learned in very high dimensional spaces, with good generalization guarantees hold as long as the *margins* on training examples are large. Margins are closely related to norms of parameter vectors.)

Finding the optimal parameters $\underline{w}^* = \arg\max_{\underline{w}} L(\underline{w})$ can again be achieved using gradient-based methods (e.g., LBFGS). The partial derivatives are again easy to compute, and are slightly modified from before:

$$\frac{\partial}{\partial w_j} L(\underline{w}) = \sum_i \phi_j(x_i, y_i) - \sum_i \sum_y p(y|x_i;\underline{w})\phi_j(x_i, y) - \lambda w_j$$

3 MEMMs

We'll now return to sequence labeling tasks, and describe *maximum-entropy Markov models* (MEMMs), which make direct use of log-linear models. In the previous lecture we introduced HMMs as a model for sequence labeling problems. MEMMs will be a useful alternative to HMMs.

Our goal will be to model the conditional distribution

$$p(s_1, s_2 \ldots s_m | x_1 \ldots x_m)$$

where each x_j for $j = 1 \ldots m$ is the j'th input symbol (for example the j'th word in a sentence), and each s_j for $j = 1 \ldots m$ is the j'th state. We'll use $\mathcal{S}$ to denote the set of possible states; we assume that $\mathcal{S}$ is a finite set.

[2] More formally, from a Bayesian standpoint the regularization term can be viewed as $\log p(\underline{w})$ where $p(\underline{w})$ is a prior (specifically, $p(\underline{w})$ is a Gaussian prior): the parameter estimates $\underline{w}^*$ are then MAP estimates. From a frequentist standpoint there have been a number of important results showing that finding parameters with a low norm leads to better generalization guarantees (i.e., better guarantees of generalization to new, test examples).

For example, in part-of-speech tagging of English, $\mathcal{S}$ would be the set of all possible parts of speech in English (noun, verb, determiner, preposition, etc.). Given a sequence of words $x_1 \dots x_m$, there are k^m possible part-of-speech sequences $s_1 \dots s_m$, where $k = |\mathcal{S}|$ is the number of possible parts of speech. We'd like to estimate a distribution over these k^m possible sequences.

In a first step, MEMMs use the following decomposition:

$$p(s_1, s_2 \dots s_m | x_1 \dots x_m) = \prod_{i=1}^{m} p(s_i | s_1 \dots s_{i-1}, x_1 \dots x_m) \quad (1)$$

$$= \prod_{i=1}^{m} p(s_i | s_{i-1}, x_1 \dots x_m) \quad (2)$$

The first equality is exact (it follows by the chain rule of conditional probabilities). The second equality follows from an *independence assumption*, namely that for all i,

$$p(s_i | s_1 \dots s_{i-1}, x_1 \dots x_m) = p(s_i | s_{i-1}, x_1 \dots x_m)$$

Hence we are making an assumption here that is similar to the Markov assumption in HMMs, i.e., that the state in the i'th position depends only on the state in the $(i-1)$'th position.

Having made these independence assumptions, we then model each term using a log-linear model:

$$p(s_i | s_{i-1}, x_1 \dots x_m) = \frac{\exp\left(\underline{w} \cdot \underline{\phi}(x_1 \dots x_m, i, s_{i-1}, s_i)\right)}{\sum_{s' \in \mathcal{S}} \exp\left(\underline{w} \cdot \underline{\phi}(x_1 \dots x_m, i, s_{i-1}, s')\right)}$$

Here $\underline{\phi}(x_1 \dots x_m, i, s, s')$ is a feature vector where:

- $x_1 \dots x_m$ is the entire sentence being tagged
- i is the position to be tagged (can take any value from 1 to m)
- s is the previous state value (can take any value in $\mathcal{S}$)
- s' is the new state value (can take any value in $\mathcal{S}$)

See the lecture slides on log-linear models (from Lecture 1) to see examples of features used in applications such as part-of-speech tagging.

Once we've defined the feature vectors $\underline{\phi}$, we can train the parameters $\underline{w}$ of the model in the usual way for log-linear models. The training examples will consist of

sentences $x_1 \dots x_m$ annotated with state sequences $s_1 \dots s_m$. Once we've trained the parameters we will have a model of

$$p(s_i | s_{i-1}, x_1 \dots x_m)$$

and hence a model of

$$p(s_1 \dots s_m | x_1 \dots x_m)$$

The next question will be how to *decode* with the model.

Decoding with MEMMs. The decoding problem is as follows. We're given a new test sequence $x_1 \dots x_m$. Our goal is to compute the most likely state sequence for this test sequence,

$$\arg \max_{s_1 \dots s_m} p(s_1 \dots s_m | x_1 \dots x_m)$$

There are k^m possible state sequences, so for any reasonably large sentence length m brute-force search through all the possibilities will not be possible.

Fortunately, we will be able to again make use of the Viterbi algorithm: it will take a very similar form to the Viterbi algorithm for HMMs. The basic data structure in the algorithm will be a dynamic programming table π with entries

$$\pi[j, s]$$

for $j = 1 \dots m$, and $s \in \mathcal{S}$. $\pi[j, s]$ will store the maximum probability for any state sequence ending in state s at position j. More formally, our algorithm will compute

$$\pi[j, s] = \max_{s_1 \dots s_{j-1}} \left(p(s | s_{j-1}, x_1 \dots x_m) \prod_{k=1}^{j-1} p(s_k | s_{k-1}, x_1 \dots x_m) \right)$$

for all $j = 1 \dots m$, and for all $s \in \mathcal{S}$.

The algorithm is as follows:

- Initialization: for $s \in \mathcal{S}$

  $$\pi[1, s] = p(s | s_0, x_1 \dots x_m)$$

  where s_0 is a special "initial" state.

- For $j = 2 \dots m$, $s = 1 \dots k$:

  $$\pi[j, s] = \max_{s' \in \mathcal{S}} \left[\pi[j-1, s'] \times p(s | s', x_1 \dots x_m) \right]$$

Finally, having filled in the $\pi[j, s]$ values for all j, s, we can calculate

$$\max_{s_1 \ldots s_m} p(s_1 \ldots s_m | x_1 \ldots x_m) = \max_s \pi[m, s]$$

The algorithm runs in $O(mk^2)$ time (i.e., linear in the sequence length m, and quadratic in the number of states k). As in the Viterbi algorithm for HMMs, we can compute the highest-scoring sequence using backpointers in the dynamic programming algorithm (see the HMM slides from lecture 1).

Comparison between MEMMs and HMMs So what is the motivation for using MEMMs instead of HMMs? Note that the Viterbi decoding algorithms for the two models are very similar. In MEMMs, the probability associated with each state transition s_{i-1} to s_i is

$$p(s_i | s_{i-1}, x_1 \ldots x_m) = \frac{\exp\left(\underline{w} \cdot \underline{\phi}(x_1 \ldots x_m, i, s_{i-1}, s_i)\right)}{\sum_{s' \in \mathcal{S}} \exp\left(\underline{w} \cdot \underline{\phi}(x_1 \ldots x_m, i, s_{i-1}, s')\right)}$$

In HMMs, the probability associated with each transition is

$$p(s_i | s_{i-1}) p(x_i | s_i)$$

The key advantage of MEMMs is that the use of feature vectors $\underline{\phi}$ allows much richer representations than those used in HMMs. For example, the transition probability can be sensitive to *any* word in the input sequence $x_1 \ldots x_m$. In addition, it is very easy to introduce features that are sensitive to spelling features (e.g., prefixes or suffixes) of the current word x_i, or of the surrounding words. These features are useful in many NLP applications, and are difficult to incorporate within HMMs in a clean way.

4 CRFs

We now turn to conditional random fields (CRFs).

One brief note on notation: for convenience, we'll use $\underline{x}$ to refer to an input sequence $x_1 \ldots x_m$, and $\underline{s}$ to refer to a sequence of states $s_1 \ldots s_m$. The set of all possible states is again $\mathcal{S}$; the set of all possible state sequences is $\mathcal{S}^m$. In conditional random fields we'll again build a model of

$$p(s_1 \ldots s_m | x_1 \ldots x_m) = p(\underline{s} | \underline{x})$$

A first key idea in CRFs will be to define a feature vector

$$\underline{\Phi}(\underline{x}, \underline{s}) \in \mathbb{R}^d$$

that maps an *entire input sequence* $\underline{x}$ *paired with an entire state sequence* $\underline{s}$ to some d-dimensional feature vector. We'll soon give a concrete definition for $\underline{\Phi}$, but for now just assume that some definition exists. We will often refer to $\underline{\Phi}$ as being a "global" feature vector (it is global in the sense that it takes the entire state sequence into account).

We then build a *giant* log-linear model,

$$p(\underline{s}|\underline{x};\underline{w}) = \frac{\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s})\right)}{\sum_{\underline{s}'\in\mathcal{S}^m}\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s}')\right)}$$

This is "just" another log-linear model, but it is is "giant" in the sense that: 1) the space of possible values for $\underline{s}$, i.e., $\mathcal{S}^m$, is huge. 2) The normalization constant (denominator in the above expression) involves a sum over the set $\mathcal{S}^m$. At first glance, these issues might seem to cause severe computational problems, but we'll soon see that under appropriate assumptions we can train and decode efficiently with this type of model.

The next question is how to define $\underline{\Phi}(\underline{x},\underline{s})$? Our answer will be

$$\underline{\Phi}(\underline{x},\underline{s}) = \sum_{j=1}^{m}\underline{\phi}(\underline{x},j,s_{j-1},s_j)$$

where $\underline{\phi}(\underline{x},j,s_{j-1},s_j)$ are the same as the feature vectors used in MEMMs. Or put another way, we're assuming that for $k = 1\ldots d$, the k'th global feature is

$$\Phi_k(\underline{x},\underline{s}) = \sum_{j=1}^{m}\phi_k(\underline{x},j,s_{j-1},s_j)$$

Thus Φ_k is calculated by summing the "local" feature vector ϕ_k over the m different state transitions in $s_1\ldots s_m$.

We now turn to two critical practical issues in CRFs: first, *decoding*, and second, *parameter estimation*.

Decoding with CRFs The decoding problem in CRFs is as follows: for a given input sequence $\underline{x} = x_1, x_2, \ldots x_m$, we would like to find the most likely underlying state sequence under the model, that is,

$$\arg\max_{\underline{s}\in\mathcal{S}^m} p(\underline{s}|\underline{x};\underline{w})$$

We simplify this expression as follows:

$$\arg\max_{\underline{s}\in\mathcal{S}^m} p(\underline{s}|\underline{x};\underline{w}) \quad = \quad \arg\max_{\underline{s}\in\mathcal{S}^m} \quad \frac{\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s})\right)}{\sum_{\underline{s}'\in\mathcal{S}^m}\exp\left(\underline{w}\cdot\underline{\Phi}(\underline{x},\underline{s}')\right)}$$

$$
\begin{aligned}
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \exp\left(\underline{w} \cdot \underline{\Phi}(\underline{x}, \underline{s})\right) \\
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \underline{w} \cdot \underline{\Phi}(\underline{x}, \underline{s}) \\
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \underline{w} \cdot \sum_{j=1}^{m} \underline{\phi}(\underline{x}, j, s_{j-1}, s_j) \\
&= \arg\max_{\underline{s} \in \mathcal{S}^m} \quad \sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j)
\end{aligned}
$$

So we have shown that finding the most likely sequence under the model is equivalent to finding the sequence that maximizes

$$\arg\max_{\underline{s} \in \mathcal{S}^m} \quad \sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j)$$

This problem has a clear intuition. Each transition from state s_{j-1} to state s_j has an associated score

$$\underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j)$$

This score could be positive or negative. Intuitively, this score will be relatively high if the state transition is plausible, relatively low if this transition is implausible. The decoding problem is to find an entire *sequence* of states such that the sum of transition scores is maximized.

We can again solve this problem using a variant of the Viterbi algorithm, in a very similar way to the decoding algorithm for HMMs or MEMMs:

- Initialization: for $s \in \mathcal{S}$

  $$\pi[1, s] = \underline{w} \cdot \underline{\phi}(\underline{x}, 1, s_0, s)$$

  where s_0 is a special "initial" state.

- For $j = 2 \ldots m$, $s = 1 \ldots k$:

  $$\pi[j, s] = \max_{s' \in \mathcal{S}} \left[\pi[j-1, s'] + \underline{w} \cdot \underline{\phi}(\underline{x}, j, s', s)\right]$$

We then have

$$\max_{s_1 \ldots s_m} \sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(\underline{x}, j, s_{j-1}, s_j) = \max_s \pi[m, s]$$

As before, backpointers can be used to allow us to recover the highest scoring state sequence. The algorithm again runs in $O(mk^2)$ time. Hence we have shown that decoding in CRFs is efficient.

Parameter Estimation in CRFs. For parameter estimation, we assume we have a set of n labeled examples, $\{(\underline{x}^i, \underline{s}^i)\}_{i=1}^n$. Each $\underline{x}^i$ is an input sequence $x_1^i \dots x_m^i$, each $\underline{s}^i$ is a state sequence $s_1^i \dots s_m^i$. We then proceed in exactly the same way as for regular log-linear models. The *regularized log-likelihood function* is

$$L(\underline{w}) = \sum_{i=1}^{n} \log p(\underline{s}^i | \underline{x}^i; \underline{w}) - \frac{\lambda}{2} ||\underline{w}||^2$$

Our parameter estimates are then

$$\underline{w}^* = \arg\max_{\underline{w} \in \mathbb{R}^d} \sum_{i=1}^{n} \log p(\underline{s}^i | \underline{x}^i; \underline{w}) - \frac{\lambda}{2} ||\underline{w}||^2$$

We'll again use gradient-based optimization methods to find $\underline{w}^*$. As before, the partial derivatives are

$$\frac{\partial}{\partial w_k} L(\underline{w}) = \sum_i \Phi_k(\underline{x}^i, \underline{s}^i) - \sum_i \sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \Phi_k(\underline{x}^i, \underline{s}) - \lambda w_k$$

The first term is easily computed, because

$$\sum_i \Phi_k(\underline{x}^i, \underline{s}^i) = \sum_i \sum_{j=1}^{m} \phi_k(\underline{x}^i, j, s_{j-1}^i, s_j^i)$$

Hence all we have to do is to sum over all training examples $i = 1 \dots n$, and for each example sum over all positions $j = 1 \dots m$.

The second term is more difficult to deal with, because it involves a sum over $\mathcal{S}^m$, a very large set. However, we will see that this term can be computed efficiently using dynamic programming. The derivation is as follows:

$$\sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \Phi_k(\underline{x}^i, \underline{s}) \tag{3}$$

$$= \sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \sum_{j=1}^{m} \phi_k(\underline{x}^i, j, s_{j-1}, s_j) \tag{4}$$

$$= \sum_{j=1}^{m} \sum_{\underline{s} \in \mathcal{S}^m} p(\underline{s} | \underline{x}^i; \underline{w}) \phi_k(\underline{x}^i, j, s_{j-1}, s_j) \tag{5}$$

$$= \sum_{j=1}^{m} \sum_{a \in \mathcal{S}, b \in \mathcal{S}} \sum_{\substack{\underline{s} \in \mathcal{S}^m : \\ s_{j-1} = a, s_j = b}} p(\underline{s} | \underline{x}^i; \underline{w}) \phi_k(\underline{x}^i, j, s_{j-1}, s_j) \tag{6}$$

$$= \sum_{j=1}^{m} \sum_{a \in \mathcal{S}, b \in \mathcal{S}} \phi_k(\underline{x}^i, j, a, b) \sum_{\substack{\underline{s} \in \mathcal{S}^m: \\ s_{j-1}=a, s_j=b}} p(\underline{s}|\underline{x}^i; \underline{w}) \tag{7}$$

$$= \sum_{j=1}^{m} \sum_{a \in \mathcal{S}, b \in \mathcal{S}} q_j^i(a, b) \phi_k(\underline{x}^i, j, a, b) \tag{8}$$

where

$$q_j^i(a, b) = \sum_{\underline{s} \in \mathcal{S}^m : s_{j-1}=a, s_j=b} p(\underline{s}|\underline{x}^i; \underline{w})$$

The important thing to note is that if we can compute the $q_j^i(a, b)$ terms efficiently, we can compute the derivatives efficiently, using the expression in Eq. 8. The quantity $q_j^i(a, b)$ has a fairly intuitive interpretation: it is the probabilty of the i'th training example $\underline{x}^i$ having state a at position $j - 1$ and state b at position j, under the distribution $p(\underline{s}|\underline{x}; \underline{w})$.

A critical result is that for a given i, *all $q_j^i(a, b)$ terms can be calculated together, in $O(mk^2)$ time*. The algorithm that achieves this is the *forward-backward algorithm*. This is another dynamic programming algorithm, and is closely related to the Viterbi algorithm.

Log-Linear Models

Michael Collins

1 Introduction

This note describes *log-linear models*, which are very widely used in natural language processing. A key advantage of log-linear models is their flexibility: as we will see, they allow a very rich set of features to be used in a model, arguably much richer representations than the simple estimation techniques we have seen earlier in the course (e.g., the smoothing methods that we initially introduced for language modeling, and which were later applied to other models such as HMMs for tagging, and PCFGs for parsing). In this note we will give motivation for log-linear models, give basic definitions, and describe how parameters can be estimated in these models. In subsequent classes we will see how these models can be applied to a number of natural language processing problems.

2 Motivation

As a motivating example, consider again the language modeling problem, where the task is to derive an estimate of the conditional probability

$$P(W_i = w_i | W_1 = w_1 \dots W_{i-1} = w_{i-1}) = p(w_i | w_1 \dots w_{i-1})$$

for any sequence of words $w_1 \dots w_i$, where i can be any positive integer. Here w_i is the i'th word in a document: our task is to model the distribution over the word w_i, conditioned on the previous sequence of words $w_1 \dots w_{i-1}$.

In trigram language models, we assumed that

$$p(w_i | w_1 \dots w_{i-1}) = q(w_i | w_{i-2}, w_{i-1})$$

where $q(w|u, v)$ for any trigram (u, v, w) is a parameter of the model. We studied a variety of ways of estimating the q parameters; as one example, we studied linear interpolation, where

$$q(w|u, v) = \lambda_1 q_{ML}(w|u, v) + \lambda_2 q_{ML}(w|v) + \lambda_3 q_{ML}(w) \quad (1)$$

Here each q_{ML} is a maximum-likelihood estimate, and $\lambda_1, \lambda_2, \lambda_3$ are parameters dictating the weight assigned to each estimate (recall that we had the constraints that $\lambda_1 + \lambda_2 + \lambda_3 = 1$, and $\lambda_i \geq 0$ for all i).

Trigram language models are quite effective, but they make relatively narrow use of the context $w_1 \ldots w_{i-1}$. Consider, for example, the case where the context $w_1 \ldots w_{i-1}$ is the following sequence of words:

> Third, the notion "grammatical in English" cannot be identified in any way with the notion "high order of statistical approximation to English". It is fair to assume that neither sentence (1) nor (2) (nor indeed any part of these sentences) has ever occurred in an English discourse. Hence, in any statistical

Assume in addition that we'd like to estimate the probability of the word *model* appearing as word w_i, i.e., we'd like to estimate

$$P(W_i = \text{model} | W_1 = w_1 \ldots W_{i-1} = w_{i-1})$$

In addition to the previous two words in the document (as used in trigram language models), we could imagine conditioning on all kinds of features of the context, which might be useful evidence in estimating the probability of seeing *model* as the next word. For example, we might consider the probability of *model* conditioned on word w_{i-2}, ignoring w_{i-1} completely:

$$P(W_i = \text{model} | W_{i-2} = \text{any})$$

We might condition on the fact that the previous word is an adjective

$$P(W_i = \text{model} | \text{pos}(W_{i-1}) = \text{adjective})$$

here pos is a function that maps a word to its part of speech. (For simplicity we assume that this is a deterministic function, i.e., the mapping from a word to its underlying part-of-speech is unambiguous.) We might condition on the fact that the previous word's suffix is "ical":

$$P(W_i = \text{model} | \text{suff4}(W_{i-1}) = \text{ical})$$

(here suff4 is a function that maps a word to its last four characters). We might condition on the fact that the word *model* does not appear in the context:

$$P(W_i = \text{model} | W_j \neq \text{model} \ \text{ for } j \in \{1 \ldots (i-1)\})$$

or we might condition on the fact that the word *grammatical* does appear in the context:

$$P(W_i = \text{model} | W_j = \text{grammatical} \text{ for some } j \in \{1 \dots (i-1)\})$$

In short, all kinds of information in the context might be useful in estimating the probability of a particular word (e.g., *model*) in that context.

A naive way to use this information would be to simply extend the methods that we saw for trigram language models. Rather than combining three estimates, based on trigram, bigram, and unigram estimates, we would combine a much larger set of estimates. We would again estimate λ parameters reflecting the importance or weight of each estimate. The resulting estimator would take something like the following form (this is intended as a sketch only):

$$\begin{aligned}
&p(\text{model} | w_1, \dots w_{i-1}) = \\
&\lambda_1 \times q_{ML}(\text{model} | w_{i-2} = \text{any}, w_{i-1} = \text{statistical}) + \\
&\lambda_2 \times q_{ML}(\text{model} | w_{i-1} = \text{statistical}) + \\
&\lambda_3 \times q_{ML}(\text{model}) + \\
&\lambda_4 \times q_{ML}(\text{model} | w_{i-2} = \text{any}) + \\
&\lambda_5 \times q_{ML}(\text{model} | w_{i-1} \text{ is an adjective}) + \\
&\lambda_6 \times q_{ML}(\text{model} | w_{i-1} \text{ ends in ``ical''}) + \\
&\lambda_7 \times q_{ML}(\text{model} | \text{``model'' does not occur somewhere in } w_1, \dots w_{i-1}) + \\
&\lambda_8 \times q_{ML}(\text{model} | \text{``grammatical'' occurs somewhere in } w_1, \dots w_{i-1}) + \\
&\dots
\end{aligned}$$

The problem is that the linear interpolation approach becomes extremely unwieldy as we add more and more pieces of conditioning information. In practice, it is very difficult to extend this approach beyond the case where we small number of estimates that fall into a natural hierarchy (e.g., unigram, bigram, trigram estimates). In contrast, we will see that log-linear models offer a much more satisfactory method for incorporating multiple pieces of contextual information.

3 A Second Example: Part-of-speech Tagging

Our second example concerns part-of-speech tagging. Consider the problem where the context is a sequence of words $w_1 \dots w_n$, together with a sequence of tags, $t_1 \dots t_{i-1}$ (here $i < n$), and our task is to model the conditional distribution over the i'th tag in the sequence. That is, we wish to model the conditional distribution

$$P(T_i = t_i | T_1 = t_1 \dots T_{i-1} = t_{i-1}, W_1 = w_1 \dots W_n = w_n)$$

As an example, we might have the following context:

> Hispaniola/NNP quickly/RB became/VB an/DT important/JJ base from which Spain expanded its empire into the rest of the Western Hemisphere .

Here $w_1 \ldots w_n$ is the sentence *Hispaniola quickly ... Hemisphere* ., and the previous sequence of tags is $t_1 \ldots t_5$ = `NNP RB VB DT JJ`. We have $i = 6$, and our task is to model the distribution

$$P(T_6 = t_6 \quad | \quad W_1 \ldots W_n = \textit{Hispaniola quickly} \ldots \textit{Hemisphere} ., \\ T_1 \ldots T_5 = \texttt{NNP RB VB DT JJ})$$

i.e., our task is to model the distribution over tags for the 6th word, *base*, in the sentence.

In this case there are again many pieces of contextual information that might be useful in estimating the distribution over values for t_i. To be concrete, consider estimating the probability that the tag for *base* is `V` (i.e., $T_6 = \texttt{V}$). We might consider the probability conditioned on the identity of the i'th word:

$$P(T_6 = \texttt{V} | W_6 = \textit{base})$$

and we might also consider the probability conditioned on the previous one or two tags:

$$P(T_6 = \texttt{V} | T_5 = \texttt{JJ})$$

$$P(T_6 = \texttt{V} | T_4 = \texttt{DT}, T_5 = \texttt{JJ})$$

We might consider the probability conditioned on the *previous* word in the sentence

$$P(T_6 = \texttt{V} | W_5 = \textit{important})$$

or the probability conditioned on the *next* word in the sentence

$$P(T_6 = \texttt{V} | W_7 = \textit{from})$$

We might also consider the probability based on spelling features of the word w_6, for example the last two letters of w_6:

$$P(T_6 = \texttt{V} | \text{suff2}(W_6) = \textit{se})$$

(here suff2 is a function that maps a word to its last two letters).

In short, we again have a scenario where a whole variety of contextual features might be useful in modeling the distribution over the random variable of interest (in this case the identity of the i'th tag). Again, a naive approach based on an extension of linear interpolation would unfortunately fail badly when faced with this estimation problem.

4 Log-Linear Models

We now describe how log-linear models can be applied to problems of the above form.

4.1 Basic Definitions

The abstract problem is as follows. We have some set of possible *inputs*, $\mathcal{X}$, and a set of possible *labels*, $\mathcal{Y}$. Our task is to model the conditional probability

$$p(y|x)$$

for any pair (x, y) such that $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.

For example, in the language modeling task we have some finite set of possible words in the language, call this set $\mathcal{V}$. The set $\mathcal{Y}$ is simply equal to $\mathcal{V}$. The set $\mathcal{X}$ is the set of possible sequences $w_1 \dots w_{i-1}$ such that $i \geq 1$, and $w_j \in \mathcal{V}$ for $j \in \{1 \dots (i-1)\}$.

In the part-of-speech tagging example, we have some set $\mathcal{V}$ of possible words, and a set $\mathcal{T}$ of possible tags. The set $\mathcal{Y}$ is simply equal to $\mathcal{T}$. The set $\mathcal{X}$ is the set of contexts of the form

$$\langle w_1 w_2 \dots w_n, t_1 t_2 \dots t_{i-1} \rangle$$

where $n \geq 1$ is an integer specifying the length of the input sentence, $w_j \in \mathcal{V}$ for $j \in \{1 \dots n\}$, $i \in \{1 \dots (n-1)\}$, and $t_j \in \mathcal{T}$ for $j \in \{1 \dots (i-1)\}$.

We will assume throughout that $\mathcal{Y}$ is a finite set. The set $\mathcal{X}$ could be finite, countably infinite, or even uncountably infinite.

Log-linear models are then defined as follows:

Definition 1 (Log-linear Models) *A log-linear model consists of the following components:*

- *A set $\mathcal{X}$ of possible inputs.*
- *A set $\mathcal{Y}$ of possible labels. The set $\mathcal{Y}$ is assumed to be finite.*
- *A positive integer d specifying the number of features and parameters in the model.*
- *A function $f : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^d$ that maps any (x, y) pair to a* feature-vector *$f(x, y)$.*
- *A parameter vector $v \in \mathbb{R}^d$.*

For any $x \in \mathcal{X}$, $y \in \mathcal{Y}$, the model defines a condtional probability

$$p(y|x; v) = \frac{\exp(v \cdot f(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(v \cdot f(x, y'))}$$

Here $\exp(x) = e^x$, and $v \cdot f(x, y) = \sum_{k=1}^{d} v_k f_k(x, y)$ is the inner product between v and $f(x, y)$. The term $p(y|x; v)$ is intended to be read as "the probability of y conditioned on x, under parameter values v". □

We now describe the components of the model in more detail, first focusing on the feature-vector definitions $f(x, y)$, then giving intuition behind the model form

$$p(y|x; v) = \frac{\exp(v \cdot f(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(v \cdot f(x, y'))}$$

5 Features

As described in the previous section, for any pair (x, y), $f(x, y) \in \mathbb{R}^d$ is a feature vector representing that pair. Each component $f_k(x, y)$ for $k = 1 \ldots d$ in this vector is referred to as a *feature*. The features allows us to represent different properties of the input x, in conjunction with the label y. Each feature has an associated parameter, v_k, whose value is estimated using a set of training examples. The training set consists of a sequence of examples $(x^{(i)}, y^{(i)})$ for $i = 1 \ldots n$, where each $x^{(i)} \in \mathcal{X}$, and each $y^{(i)} \in \mathcal{Y}$.

In this section we first give an example of how features can be constructed for the language modeling problem, as introduced earlier in this note; we then describe some practical issues in defining features.

5.1 Features for the Language Modeling Example

Consider again the language modeling problem, where the input x is a sequence of words $w_1 w_2 \ldots w_{i-1}$, and the label y is a word. Figure 1 shows a set of example features for this problem. Each feature is an indicator function: that is, each feature is a function that returns either 1 or 0. It is extremely common in NLP applications to have indicator functions as features. Each feature returns the value of 1 if some property of the input x conjoined with the label y is true, and 0 otherwise.

The first three features, f_1, f_2, and f_3, are analogous to unigram, bigram, and trigram features in a regular trigram language model. The first feature returns 1 if the label y is equal to the word *model*, and 0 otherwise. The second feature returns 1 if the bigram $\langle w_{i-1}\ y \rangle$ is equal to $\langle$*statistical model*$\rangle$, and 0 otherwise. The third feature returns 1 if the trigram $\langle w_{i-2}\ w_{i-1}\ y \rangle$ is equal to $\langle$*the statistical model*$\rangle$,

$$
\begin{aligned}
f_1(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model} \\ 0 & \text{otherwise} \end{cases} \\
f_2(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model} \text{ and } w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases} \\
f_3(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any}, w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases} \\
f_4(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any} \\ 0 & \text{otherwise} \end{cases} \\
f_5(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-1} \text{ is an adjective} \\ 0 & \text{otherwise} \end{cases} \\
f_6(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-1} \text{ ends in ``ical''} \\ 0 & \text{otherwise} \end{cases} \\
f_7(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, \text{``model'' is not in } w_1, \ldots w_{i-1} \\ 0 & \text{otherwise} \end{cases} \\
f_8(x,y) &= \begin{cases} 1 & \text{if } y = \texttt{model}, \text{``grammatical'' is in } w_1, \ldots w_{i-1} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 1: Example features for the language modeling problem, where the input x is a sequence of words $w_1 w_2 \ldots w_{i-1}$, and the label y is a word.

and 0 otherwise. Recall that each of these features will have a parameter, v_1, v_2, or v_3; these parameters will play a similar role to the parameters in a regular trigram language model.

The features $f_4 \ldots f_8$ in figure 1 consider properties that go beyond unigram, bigram, and trigram features. The feature f_4 considers word w_{i-2} in conjunction with the label y, ignoring the word w_{i-1}; this type of feature is often referred to as a "skip bigram". Feature f_5 considers the part-of-speech of the previous word (assume again that the part-of-speech for the previous word is available, for example through a deterministic mapping from words to their part-of-speech, or perhaps through a POS tagger's output on words $w_1 \ldots w_{i-1}$). Feature f_6 considers the suffix of the previous word, and features f_7 and f_8 consider various other features of the input $x = w_1 \ldots w_{i-1}$.

From this example we can see that it is possible to incorporate a broad set of contextual information into the language modeling problem, using features which are indicator functions.

5.2 Feature Templates

We now discuss some practical issues in defining features. In practice, a key idea in defining features is that of *feature templates*. We introduce this idea in this section.

Recall that our first three features in the previous example were as follows:

$$f_1(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model} \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model} \text{ and } w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases}$$

$$f_3(x, y) = \begin{cases} 1 & \text{if } y = \texttt{model}, w_{i-2} = \texttt{any}, w_{i-1} = \texttt{statistical} \\ 0 & \text{otherwise} \end{cases}$$

These features track the unigram $\langle$*model*$\rangle$, the bigram $\langle$*statistical model*$\rangle$, and the trigram $\langle$*the statistical model*$\rangle$.

Each of these features is specific to a particular unigram, bigram or trigram. In practice, we would like to define a much larger class of features, which consider all possible unigrams, bigrams or trigrams seen in the training data. To do this, we use *feature templates* to generate large sets of features.

As one example, here is a feature template for trigrams:

Definition 2 (Trigram feature template) *For any trigram* (u, v, w) *seen in train-*

ing data, create a feature

$$f_{N(u,v,w)}(x,y) = \begin{cases} 1 & \textit{if } y = w, \, w_{i-2} = u, \, w_{i-1} = v \\ 0 & \textit{otherwise} \end{cases}$$

where $N(u,v,w)$ is a function that maps each trigram in the training data to a unique integer.

A couple of notes on this definition:

- Note that the template only generates trigram features for those trigrams seen in training data. There are two reasons for this restriction. First, it is not feasible to generate a feature for every possible trigram, even those not seen in training data: this would lead to V^3 features, where V is the number of words in the vocabulary, which is a very large set of features. Second, for any trigram (u,v,w) not seen in training data, we do not have evidence to estimate the associated parameter value, so there is no point including it in any case.[1]

- The function $N(u,v,w)$ maps each trigram to a unique integer: that is, it is a function such that for any trigrams (u,v,w) and (u',v',w') such that $u \neq u'$, $v \neq v'$, or $w \neq w'$, we have

  $$N(u,v,w) \neq N(u',v',w')$$

  In practice, in implementations of feature templates, the function N is implemented through a hash function. For example, we could use a hash table to hash strings such as `trigram=the_statistical_model` to integers. Each distinct string is hashed to a different integer.

Continuing with the example, we can also define bigram and unigram feature templates:

Definition 3 (Bigram feature template) *For any bigram (v,w) seen in training data, create a feature*

$$f_{N(v,w)}(x,y) = \begin{cases} 1 & \textit{if } y = w, \, w_{i-1} = v \\ 0 & \textit{otherwise} \end{cases}$$

where $N(v,w)$ maps each bigram to a unique integer.

[1] This isn't quite accurate: there may in fact be reasons for including features for trigrams (u,v,w) where the bigram (u,v) *is* observed in the training data, but the trigram (u,v,w) is *not* observed in the training data. We defer discussion of this until later.

Definition 4 (Unigram feature template) *For any unigram (w) seen in training data, create a feature*

$$f_{N(w)}(x, y) = \begin{cases} 1 & \text{if } y = w \\ 0 & \text{otherwise} \end{cases}$$

where $N(w)$ maps each unigram to a unique integer.

We actually need to be slightly more careful with these definitions, to avoid overlap between trigram, bigram, and unigram features. Define T, B and U to be the set of trigrams, bigrams, and unigrams seen in the training data. Define

$$N_t = \{i : \exists (u, v, w) \in T \text{ such that } N(u, v, w) = i\}$$

$$N_n = \{i : \exists (v, w) \in T \text{ such that } N(v, w) = i\}$$

$$N_u = \{i : \exists (w) \in T \text{ such that } N(w) = i\}$$

Then we need to make sure that there is no overlap between these sets—otherwise, two different n-grams would be mapped to the same feature. More formally, we need

$$N_t \cap N_b = N_t \cap N_u = N_b \cap N_u = \emptyset \tag{2}$$

In practice, it is easy to ensure this when implementing log-linear models, using a single hash table to hash strings such as `trigram=the_statistical_model`, `bigram=statistical_model`, `unigram=model`, to distinct integers.

We could of course define additional templates. For example, the following is a template which tracks the length-4 suffix of the previous word, in conjunction with the label y:

Definition 5 (Length-4 Suffix Template) *For any pair (v, w) seen in training data, where $v =$ suff4(w_{i-1}), and $w = y$, create a feature*

$$f_{N(\textit{suff4}=v,w)}(x, y) = \begin{cases} 1 & \text{if } y = w \text{ and } \textit{suff4}(x) = v \\ 0 & \text{otherwise} \end{cases}$$

where $N(\textit{suff4} = v, w)$ maps each pair (v, w) to a unique integer, with no overlap with the other feature templates used in the model (where overlap is defined analogously to Eq. 2 above).

5.3 Feature Sparsity

A very important property of the features we have defined above is feature sparsity. The number of features, d, in many NLP applications can be extremely large. For example, with just the trigram template defined above, we would have one feature for each trigram seen in training data. It is not untypical to see models with 100s of thousands or even millions of features.

This raises obvious concerns with efficiency of the resulting models. However, we describe in this section how feature sparsity can lead to efficient models.

The key observation is the following: for any given pair (x, y), the number of values for k in $\{1 \dots d\}$ such that

$$f_k(x, y) = 1$$

is often very small, and is typically much smaller than the total number of features, d. Thus all but a very small subset of the features are 0: the feature vector $f(x, y)$ is a very sparse bit-string, where almost all features $f_k(x, y)$ are equal to 0, and only a few features are equal to 1.

As one example, consider the language modeling example where we use only the trigram, bigram and unigram templates, as described above. The number of features in this model is large (it is equal to the number of distinct trigrams, bigrams and unigrams seen in training data). However, it can be seen immediately that for any pair (x, y), at most three features are non-zero (in the worst case, the pair (x, y) contains trigram, bigram and unigram features which are all seen in the training data, giving three non-zero features in total).

When implementing log-linear models, models with sparse features can be quite efficient, because there is no need to explicitly represent and manipulate d-dimensional feature vectors $f(x, y)$. Instead, it is generally much more efficient to implement a function (typically through hash tables) that for any pair (x, y) computes the indices of the non-zero features: i.e., a function that computes the set

$$Z(x, y) = \{k : f_k(x, y) = 1\}$$

This set is small in sparse feature spaces—for example with unigram/bigram/trigram features alone, it would be of size at most 3. In general, it is straightforward to implement a function that computes $Z(x, y)$ in $O(|Z(x, y)|)$ time, using hash functions. Note that $|Z(x, y)| \ll d$, so this is much more efficient than explicitly computing all d features, which would take $O(d)$ time.

As one example of how efficient computation of $Z(x, y)$ can be very helpful, consider computation of the inner product

$$v \cdot f(x, y) = \sum_{k=1}^{d} v_k f_k(x, y)$$

This computation is central in log-linear models. A naive method would iterate over each of the d features in turn, and would take $O(d)$ time. In contrast, if we make use of the identity

$$\sum_{k=1}^{d} v_k f_k(x, y) = \sum_{k \in Z(x,y)} v_k$$

hence looking at only non-zero features, we can compute the inner product in $O(|Z(x, y)|)$ time.

6 The Model form for Log-Linear Models

We now describe the model form for log-linear models in more detail. Recall that for any pair (x, y) such that $x \in \mathcal{X}$, and $y \in \mathcal{Y}$, the conditional probability under the model is

$$p(y \mid x; v) = \frac{\exp(v \cdot f(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(v \cdot f(x, y'))}$$

The inner products

$$v \cdot f(x, y)$$

play a key role in this expression. Again, for illustration consider our languge-modeling example where the input $x = w_1 \dots w_{i-1}$ is the following sequence of words:

> Third, the notion "grammatical in English" cannot be identified in any way with the notion "high order of statistical approximation to English". It is fair to assume that neither sentence (1) nor (2) (nor indeed any part of these sentences) has ever occurred in an English discourse. Hence, in any statistical

The first step in calculating the probability distribution over the next word in the document, conditioned on x, is to calculate the inner product $v \cdot f(x, y)$ for each possible label y (i.e., for each possible word in the vocabulary). We might, for example, find the following values (we show the values for just a few possible words—in reality we would compute an inner product for each possible word):

$$\begin{aligned} v \cdot f(x, model) &= 5.6 & v \cdot f(x, the) &= -3.2 \\ v \cdot f(x, is) &= 1.5 & v \cdot f(x, of) &= 1.3 \\ v \cdot f(x, models) &= 4.5 & \dots \end{aligned}$$

Note that the inner products can take any value in the reals, positive or negative. Intuitively, if the inner product $v \cdot f(x, y)$ for a given word y is high, this indicates that the word has high probability given the context x. Conversely, if $v \cdot f(x, y)$ is low, it indicates that y has low probability in this context.

The inner products $v \cdot f(x, y)$ can take any value in the reals; our goal, however, is to define a conditional distribution $p(y|x)$. If we take

$$\exp\left(v \cdot f(x, y)\right)$$

for any label y, we now have a value that is greater than 0. If $v \cdot f(x, y)$ is high, this value will be high; if $v \cdot f(x, y)$ is low, for example if it is strongly negative, this value will be low (close to zero).

Next, if we divide the above quantity by

$$\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)$$

giving

$$p(y|x; v) = \frac{\exp\left(v \cdot f(x, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)} \tag{3}$$

then it is easy to verify that we have a well-formed distribution: that is,

$$\sum_{y \in \mathcal{Y}} p(y|x; v) = 1$$

Thus the denominator in Eq. 3 is a normalization term, which ensures that we have a distribution that sums to one. In summary, the function

$$\frac{\exp\left(v \cdot f(x, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)}$$

performs a transformation which takes as input a set of values $\{v \cdot f(x, y) : y \in \mathcal{Y}\}$, where each $v \cdot f(x, y)$ can take any value in the reals, and as output produces a probability distribution over the labels $y \in \mathcal{Y}$.

Finally, we consider where the name log-linear models originates from. It follows from the above definitions that

$$\begin{aligned} \log p(y|x; v) &= v \cdot f(x, y) - \log \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right) \\ &= v \cdot f(x, y) - g(x) \end{aligned}$$

where

$$g(x) = \log \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x, y')\right)$$

The first term, $v \cdot f(x, y)$, is linear in the features $f(x, y)$. The second term, $g(x)$, depends only on x, and does not depend on the label y. Hence the log probability $\log p(y|x; v)$ is a linear function in the features $f(x, y)$, as long as we hold x fixed; this justifies the term "log-linear".

7 Parameter Estimation in Log-Linear Models

7.1 The Log-Likelihood Function, and Regularization

We now consider the problem of parameter estimation in log-linear models. We assume that we have a training set, consisting of examples $(x^{(i)}, y^{(i)})$ for $i \in \{1 \dots n\}$, where each $x^{(i)} \in \mathcal{X}$, and each $y^{(i)} \in \mathcal{Y}$.

Given parameter values v, for any example i, we can calculate the log conditional probability

$$\log p(y^{(i)}|x^{(i)}; v)$$

under the model. Intuitively, the higher this value, the better the model fits this particular example. The log-likelihood considers the sum of log probabilities of examples in the training data:

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) \tag{4}$$

This is a function of the parameters v. For any parameter vector v, the value of $L(v)$ can be interpreted of a measure of how well the parameter vector fits the training examples.

The first estimation method we will consider is maximum-likelihood estimation, where we choose our parameters as

$$v_{ML} = \arg\max_{v \in \mathbb{R}^d} L(v)$$

In the next section we describe how the parameters v_{ML} can be found efficiently. Intuitively, this estimation method finds the parameters which fit the data as well as possible.

The maximum-likelihood estimates can run into problems, in particular in cases where the number of features in the model is very large. To illustrate, consider the language-modeling problem again, and assume that we have trigram, bigram and unigram features. Now assume that we have some trigram (u, v, w) which is seen only once in the training data; to be concrete, assume that the trigram is *the statistical model*, and assume that this trigram is seen on the 100'th

training example alone. More precisely, we assume that

$$f_{N(the,statistical,model)}(x^{(100)}, y^{(100)}) = 1$$

In addition, assume that this is the only trigram (u, v, w) in training data with $u =$ *the*, and $v =$ *statistical*. In this case, it can be shown that the maximum-likelihood parameter estimate for v_{100} is $+\infty$,[2], which gives

$$p(y^{(100)}|x^{(100)}; v) = 1$$

In fact, we have a very similar situation to the case in maximum-likelihood estimates for regular trigram models, where we would have

$$q_{ML}(\textit{model}|\textit{the, statistical}) = 1$$

for this trigram. As discussed earlier in the class, this model is clearly under-smoothed, and it will generalize badly to new test examples. It is unreasonable to assign

$$P(W_i = \textit{model}|W_{i-1}, W_{i-2} = \textit{the, statistical}) = 1$$

based on the evidence that the bigram *the statistical* is seen once, and on that one instance the bigram is followed by the word *model*.

A very common solution for log-linear models is to modify the objective function in Eq. 4 to include a *regularization term*, which prevents parameter values from becoming too large (and in particular, prevents parameter values from diverging to infinity). A common regularization term is the 2-norm of the parameter values, that is,

$$||v||^2 = \sum_k v_k^2$$

(here $||v||$ is simply the length, or Euclidean norm, of a vector v; i.e., $||v|| = \sqrt{\sum_k v_k^2}$). The modified objective function is

$$L'(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) - \frac{\lambda}{2} \sum_k v_k^2 \tag{5}$$

[2]It is relatively easy to prove that v_{100} can diverge to ∞. To give a sketch: under the above assumptions, the feature $f_{N(the,statistical,model)}(x, y)$ is equal to 1 on only a single pair $x^{(i)}, y$ where $i \in \{1 \ldots n\}$, and $y \in \mathcal{Y}$, namely the pair $(x^{(100)}, y^{(100)})$. Because of this, as $v_{100} \to \infty$, we will have $p(y^{(100)}|x^{(100)}; v)$ tending closer and closer to a value of 1, with all other values $p(y^{(i)}|x^{(i)}; v)$ remaining unchanged. Thus we can use this one parameter to maximize the value for $\log p(y^{(100)}|x^{(100)}; v)$, independently of the probability of all other examples in the training set.

where $\lambda > 0$ is a parameter, which is typically chosen by validation on some held-out dataset. We again choose the parameter values to maximize the objective function: that is, our optimal parameter values are

$$v^* = \arg\max_v L'(v)$$

The key idea behind the modified objective in Eq. 5 is that we now balance two separate terms. The first term is the log-likelihood on the training data, and can be interpreted as a measure of how well the parameters v fit the training examples. The second term is a penalty on large parameter values: it encourages parameter values to be as close to zero as possible. The parameter λ defines the relative weighting of the two terms. In practice, the final parameters v^* will be a compromise between fitting the data as well as is possible, and keeping their values as small as possible.

In practice, this use of regularization is very effective in smoothing of log-linear models.

7.2 Finding the Optimal Parameters

First, consider finding the maximum-likelihood parameter estimates: that is, the problem of finding

$$v_{ML} = \arg\max_{v \in \mathbb{R}^d} L(v)$$

where

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v)$$

The bad news is that in the general case, there is no closed-form solution for the maximum-likelihood parameters v_{ML}. The good news is that finding $\arg\max_v L(v)$ is a relatively easy problem, because $L(v)$ can be shown to be a *convex* function. This means that simple gradient-ascent-style methods will find the optimal parameters v_{ML} relatively quickly.

Figure 2 gives a sketch of a gradient-based algorithm for optimization of $L(v)$. The parameter vector is initialized to the vector of all zeros. At each iteration we first calculate the gradients δ_k for $k = 1 \ldots d$. We then move in the direction of the gradient: more precisely, we set $v \leftarrow v + \beta^* \times \delta$ where β^* is chosen to give the optimal improvement in the objective function. This is a "hill-climbing" technique where at each point we compute the steepest direction to move in (i.e., the direction of the gradient); we then move the distance in that direction which gives the greatest value for $L(v)$.

Simple gradient ascent, as shown in figure 2, can be rather slow to converge. Fortunately there are many standard packages for gradient-based optimization,

Initialization: $v = 0$

Iterate until convergence:

- Calculate $\delta_k = \frac{dL(v)}{dv_k}$ for $k = 1 \ldots d$
- Calculate $\beta^* = \arg\max_{\beta \in \mathbb{R}} L(v + \beta\delta)$ where δ is the vector with components δ_k for $k = 1 \ldots d$ (this step is performed using some type of line search)
- Set $v \leftarrow v + \beta^* \delta$

Figure 2: A gradient ascent algorithm for optimization of $L(v)$.

which use more sophisticated algorithms, and which give considerably faster convergence. As one example, a commonly used method for parameter estimation in log-linear models is LBFGs. LBFGs is again a gradient method, but it makes a more intelligent choice of search direction at each step. It does however rely on the computation of $L(v)$ and $\frac{dL(v)}{dv_k}$ for $k = 1$ at each step—in fact this is the only information it requires about the function being optimized. In summary, if we can compute $L(v)$ and $\frac{dL(v)}{dv_k}$ efficiently, then it is simple to use an existing gradient-based optimization package (e.g., based on LBFGs) to find the maximum-likelihood estimates.

Optimization of the regularized objective function,

$$L'(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) - \frac{\lambda}{2} \sum_k v_k^2$$

can be performed in a very similar manner, using gradient-based methods. $L'(v)$ is also a convex function, so a gradient-based method will find the global optimum of the parameter estimates.

The one remaining step is to describe how the gradients

$$\frac{dL(v)}{dv_k}$$

and

$$\frac{dL'(v)}{dv_k}$$

can be calculated. This is the topic of the next section.

7.3 Gradients

We first consider the derivatives

$$\frac{dL(v)}{dv_k}$$

where

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v)$$

It is relatively easy to show (see the appendix of this note), that for any $k \in \{1 \dots d\}$,

$$\frac{dL(v)}{dv_k} = \sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y) \tag{6}$$

where as before

$$p(y|x^{(i)}; v) = \frac{\exp\left(v \cdot f(x^{(i)}, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)}$$

The expression in Eq. 6 has a quite intuitive form. The first part of the expression,

$$\sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)})$$

is simply the number of times that the feature f_k is equal to 1 on the training examples (assuming that f_k is an indicator function; i.e., assuming that $f_k(x^{(i)}, y^{(i)})$ is either 1 or 0). The second part of the expression,

$$\sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y)$$

can be interpreted as the expected number of times the feature is equal to 1, where the expectation is taken with respect to the distribution

$$p(y|x^{(i)}; v) = \frac{\exp\left(v \cdot f(x^{(i)}, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)}$$

specified by the current parameters. The gradient is then the difference of these terms. It can be seen that the gradient is easily calculated.

The gradients

$$\frac{dL'(v)}{dv_k}$$

where

$$L'(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v) - \frac{\lambda}{2} \sum_k v_k^2$$

are derived in a very similar way. We have

$$\frac{d}{dv_k} \left(\sum_k v_k^2 \right) = v_k$$

hence

$$\frac{dL'(v)}{dv_k} = \sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y) - \lambda v_k \qquad (7)$$

Thus the only difference from the gradient in Eq. 6 is the additional term $-\lambda v_k$ in this expression.

A Calculation of the Derivatives

In this appendix we show how to derive the expression for the derivatives, as given in Eq. 6. Our goal is to find an expression for

$$\frac{dL(v)}{dv_k}$$

where

$$L(v) = \sum_{i=1}^{n} \log p(y^{(i)}|x^{(i)}; v)$$

First, consider a single term $\log p(y^{(i)}|x^{(i)}; v)$. Because

$$p(y^{(i)}|x^{(i)}; v) = \frac{\exp\left(v \cdot f(x^{(i)}, y^{(i)})\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)}$$

we have

$$\log p(y^{(i)}|x^{(i)}; v) = v \cdot f(x^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)$$

The derivative of the first term in this expression is simple:

$$\frac{d}{dv_k} \left(v \cdot f(x^{(i)}, y^{(i)})\right) = \frac{d}{dv_k} \left(\sum_k v_k f_k(x^{(i)}, y^{(i)}) \right) = f_k(x^{(i)}, y^{(i)}) \qquad (8)$$

Now consider the second term. This takes the form

$$\log g(v)$$

where

$$g(v) = \sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)$$

By the usual rules of differentiation,

$$\frac{d}{dv_k} \log g(v) = \frac{\frac{d}{dv_k}(g(v))}{g(v)}$$

In addition, it can be verified that

$$\frac{d}{dv_k} g(v) = \sum_{y' \in \mathcal{Y}} f_k(x^{(i)}, y') \exp\left(v \cdot f(x^{(i)}, y')\right)$$

hence

$$\begin{aligned}
\frac{d}{dv_k} \log g(v) &= \frac{\frac{d}{dv_k}(g(v))}{g(v)} && (9)\\
&= \frac{\sum_{y' \in \mathcal{Y}} f_k(x^{(i)}, y') \exp\left(v \cdot f(x^{(i)}, y')\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)} && (10)\\
&= \sum_{y' \in \mathcal{Y}} \left(f_k(x^{(i)}, y') \times \frac{\exp\left(v \cdot f(x^{(i)}, y')\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(v \cdot f(x^{(i)}, y')\right)} \right) && (11)\\
&= \sum_{y' \in \mathcal{Y}} f_k(x^{(i)}, y') p(y'|x; v) && (12)
\end{aligned}$$

Combining Eqs 8 and 12 gives

$$\frac{dL(v)}{dv_k} = \sum_{i=1}^{n} f_k(x^{(i)}, y^{(i)}) - \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} p(y|x^{(i)}; v) f_k(x^{(i)}, y)$$

The Forward-Backward Algorithm

Michael Collins

1 Introduction

This note describes the *forward-backward algorithm*. The forward-backward algorithm has very important applications to both hidden Markov models (HMMs) and conditional random fields (CRFs). It is a dynamic programming algorithm, and is closely related to the Viterbi algorithm for decoding with HMMs or CRFs.

This note describes the algorithm at a level of abstraction that applies to both HMMs and CRFs. We will also describe its specific application to these cases.

2 The Forward-Backward Algorithm

The problem set-up is as follows. Assume that we have some sequence length m, and some set of possible states $\mathcal{S}$. For any state sequence $s_1 \dots s_m$ where each $s_i \in \mathcal{S}$, we define the *potential* for the sequence as

$$\psi(s_1 \dots s_m) = \prod_{j=1}^{m} \psi(s_{j-1}, s_j, j)$$

Here we define s_0 to be *, where * is a special start symbol in the model. Here $\psi(s, s', j) \geq 0$ for $s, s' \in \mathcal{S}$, $j \in \{1 \dots m\}$ is a potential function, which returns a value for the state transition s to s' at position j in the sequence.

The potential functions $\psi(s_{j-1}, s_j, j)$ might be defined in various ways. As one example, consider an HMM applied to an input sentence $x_1 \dots x_m$. If we define

$$\psi(s', s, j) = t(s|s')e(x_j|s)$$

then

$$\begin{aligned} \psi(s_1 \dots s_m) &= \prod_{j=1}^{m} \psi(s_{j-1}, s_j, j) \\ &= \prod_{j=1}^{m} t(s_j|s_{j-1})e(x_j|s_j) \end{aligned}$$

$$= \quad p(x_1 \ldots x_m, s_1 \ldots s_m)$$

where $p(x_1 \ldots x_m, s_1 \ldots s_m)$ is the probability mass function under the HMM.

As another example, consider a CRF where we have a feature-vector definition $\underline{\phi}(x_1 \ldots x_m, s', s, j) \in \mathbb{R}^d$, and a parameter vector $\underline{w} \in \mathbb{R}^d$. Assume again that we have an input sentence $x_1 \ldots x_m$. If we define

$$\psi(s', s, j) = \exp\left(\underline{w} \cdot \underline{\phi}(x_1 \ldots x_m, s', s, j)\right)$$

then

$$\begin{aligned}
\psi(s_1 \ldots s_m) &= \prod_{j=1}^{m} \psi(s_{j-1}, s_j, j) \\
&= \prod_{j=1}^{m} \exp\left(\underline{w} \cdot \underline{\phi}(x_1 \ldots x_m, s_{j-1}, s_j, j)\right) \\
&= \exp\left(\sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(x_1 \ldots x_m, s_{j-1}, s_j, j)\right)
\end{aligned}$$

Note in particular, by the model form for CRFs, it follows that

$$p(s_1 \ldots s_m | x_1 \ldots x_m) = \frac{\psi(s_1 \ldots s_m)}{\sum_{s_1 \ldots s_m} \psi(s_1 \ldots s_m)}$$

The forward-backward algorithm is shown in figure 1. Given inputs consisting of a sequence length m, a set of possible states $\mathcal{S}$, and potential functions $\psi(s', s, j)$ for $s, s' \in \mathcal{S}$, and $j \in \{1 \ldots m\}$, it computes the following quantities:

1. $Z = \sum_{s_1 \ldots s_m} \psi(s_1 \ldots s_m)$.
2. For all $j \in \{1 \ldots m\}$, $a \in \mathcal{S}$,
$$\mu(j, a) = \sum_{s_1 \ldots s_m : s_j = a} \psi(s_1 \ldots s_m)$$
3. For all $j \in \{1 \ldots (m-1)\}$, $a, b \in \mathcal{S}$,
$$\mu(j, a, b) = \sum_{s_1 \ldots s_m : s_j = a, s_{j+1} = b} \psi(s_1 \ldots s_m)$$

Inputs: Length m, set of possible states $\mathcal{S}$, function $\psi(s, s', j)$. Define * to be a special initial state.

Initialization (forward terms): For all $s \in \mathcal{S}$,

$$\alpha(1, s) = \psi(*, s, 1)$$

Recursion (forward terms): For all $j \in \{2 \dots m\}, s \in \mathcal{S}$,

$$\alpha(j, s) = \sum_{s' \in \mathcal{S}} \alpha(j-1, s') \times \psi(s', s, j)$$

Initialization (backward terms): For all $s \in \mathcal{S}$,

$$\beta(m, s) = 1$$

Recursion (backward terms): For all $j \in \{1 \dots (m-1)\}, s \in \mathcal{S}$,

$$\beta(j, s) = \sum_{s' \in \mathcal{S}} \beta(j+1, s') \times \psi(s, s', j+1)$$

Calculations:

$$Z = \sum_{s \in \mathcal{S}} \alpha(m, s)$$

For all $j \in \{1 \dots m\}, a \in \mathcal{S}$,

$$\mu(j, a) = \alpha(j, a) \times \beta(j, a)$$

For all $j \in \{1 \dots (m-1)\}, a, b \in \mathcal{S}$,

$$\mu(j, a, b) = \alpha(j, a) \times \psi(a, b, j+1) \times \beta(j+1, b)$$

Figure 1: The forward-backward algorithm.

3 Application to CRFs

The quantities computed by the forward-backward algorithm play a central role in CRFs. First, consider the problem of calculating the conditional probability

$$p(s_1 \dots s_m | x_1 \dots x_m) = \frac{\exp\left(\sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(x_1 \dots x_m, s_{j-1}, s_j, j)\right)}{\sum_{s_1 \dots s_m} \exp\{\left(\sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(x_1 \dots x_m, s_{j-1}, s_j, j)\right)}$$

The numerator in the above expression is easy to compute; the denominator is more challenging, because it requires a sum over an exponential number of state sequences. However, if we define

$$\psi(s', s, j) = \exp\left(\underline{w} \cdot \underline{\phi}(x_1 \dots x_m, s', s, j)\right)$$

in the algorithm in figure 1, then as we argued before we have

$$\psi(s_1 \dots s_m) = \exp\left(\sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(x_1 \dots x_m, s_{j-1}, s_j, j)\right)$$

It follows that the quantity Z calculated by the algorithm is equal to the denominator in the above expression; that is,

$$Z = \sum_{s_1 \dots s_m} \exp\left(\sum_{j=1}^{m} \underline{w} \cdot \underline{\phi}(x_1 \dots x_m, s_{j-1}, s_j, j)\right)$$

Next, recall that the key difficulty in the calculation of the gradient of the log-likelihood function in CRFs was to calculate the terms

$$q_j^i(a, b) = \sum_{\underline{s}: s_{j-1}=a, s_j=b} p(\underline{s} | \underline{x}^i; \underline{w})$$

for a given input sequence $\underline{x}^i = x_1^i \dots x_m^i$, for each $j \in \{2 \dots m\}$, for each $a, b \in \mathcal{S}$ (see the note on log-linear models). Again, if we define

$$\psi(s', s, j) = \exp\left(\underline{w} \cdot \underline{\phi}(x_1^i \dots x_m^i, s', s, j)\right)$$

then it can be verified that

$$q_j^i(a, b) = \frac{\mu(j, a, b)}{Z}$$

where $\mu(j, a, b)$ and Z are the terms computed by the algorithm in figure 1.

Probabilistic Context-Free Grammars (PCFGs)

Michael Collins

1 Context-Free Grammars

1.1 Basic Definition

A context-free grammar (CFG) is a 4-tuple $G = (N, \Sigma, R, S)$ where:

- N is a finite set of non-terminal symbols.
- Σ is a finite set of terminal symbols.
- R is a finite set of rules of the form $X \rightarrow Y_1 Y_2 \ldots Y_n$, where $X \in N$, $n \geq 0$, and $Y_i \in (N \cup \Sigma)$ for $i = 1 \ldots n$.
- $S \in N$ is a distinguished start symbol.

Figure 1 shows a very simple context-free grammar, for a fragment of English. In this case the set of non-terminals N specifies some basic syntactic categories: for example S stands for "sentence", NP for "noun phrase", VP for "verb phrase", and so on. The set Σ contains the set of words in the vocabulary. The start symbol in this grammar is S: as we will see, this specifies that every parse tree has S as its root. Finally, we have context-free rules such as

$$\texttt{S} \rightarrow \texttt{NP VP}$$

or

$$\texttt{NN} \rightarrow \texttt{man}$$

The first rule specifies that an S (sentence) can be composed of an NP followed by a VP. The second rule specifies that an NN (a singular noun) can be composed of the word man.

Note that the set of allowable rules, as defined above, is quite broad: we can have any rule $X \rightarrow Y_1 \ldots Y_n$ as long as X is a member of N, and each Y_i for

$i = 1 \ldots n$ is a member of either N or Σ. We can for example have "unary rules", where $n = 1$, such as the following:

$$\begin{aligned} \texttt{NN} &\rightarrow \texttt{man} \\ \texttt{S} &\rightarrow \texttt{VP} \end{aligned}$$

We can also have rules that have a mixture of terminal and non-terminal symbols on the right-hand-side of the rule, for example

$$\begin{aligned} \texttt{VP} &\rightarrow \texttt{John Vt Mary} \\ \texttt{NP} &\rightarrow \texttt{the NN} \end{aligned}$$

We can even have rules where $n = 0$, so that there are no symbols on the right-hand-side of the rule. Examples are

$$\begin{aligned} \texttt{VP} &\rightarrow \epsilon \\ \texttt{NP} &\rightarrow \epsilon \end{aligned}$$

Here we use ϵ to refer to the empty string. Intuitively, these latter rules specify that a particular non-terminal (e.g., VP), is allowed to have no words below it in a parse tree.

1.2 (Left-most) Derivations

Given a context-free grammar G, a left-most derivation is a sequence of strings $s_1 \ldots s_n$ where

- $s_1 = \texttt{S}$. i.e., s_1 consists of a single element, the start symbol.
- $s_n \in \Sigma^*$, i.e. s_n is made up of terminal symbols only (we write Σ^* to denote the set of all possible strings made up of sequences of words taken from Σ.)
- Each s_i for $i = 2 \ldots n$ is derived from s_{i-1} by picking the left-most non-terminal X in s_{i-1} and replacing it by some β where $X \rightarrow \beta$ is a rule in R.

As one example, one left-most derivation under the grammar in figure 1 is the following:

- $s_1 = \texttt{S}$.
- $s_2 = \texttt{NP VP}$. (We have taken the left-most non-terminal in s_1, namely S, and chosen the rule $\texttt{S} \rightarrow \texttt{NP VP}$, thereby replacing S by NP followed by VP.)

N = {S, NP, VP, PP, DT, Vi, Vt, NN, IN}
S = S
Σ = {sleeps, saw, man, woman, dog, telescope, the, with, in}

$R =$

S	→	NP	VP
VP	→	Vi	
VP	→	Vt	NP
VP	→	VP	PP
NP	→	DT	NN
NP	→	NP	PP
PP	→	IN	NP

Vi	→	sleeps
Vt	→	saw
NN	→	man
NN	→	woman
NN	→	telescope
NN	→	dog
DT	→	the
IN	→	with
IN	→	in

Figure 1: A simple context-free grammar. Note that the set of non-terminals N contains a basic set of syntactic categories: S=sentence, VP=verb phrase, NP=noun phrase, PP=prepositional phrase, DT=determiner, Vi=intransitive verb, Vt=transitive verb, NN=noun, IN=preposition. The set Σ is the set of possible words in the language.

- s_3 = `DT NN VP`. (We have used the rule `NP` → `DT NN` to expand the left-most non-terminal, namely `NP`.)
- s_4 = `the NN VP`. (We have used the rule `DT` → `the`.)
- s_5 = `the man VP`. (We have used the rule `NN` → `man`.)
- s_6 = `the man Vi`. (We have used the rule `VP` → `Vi`.)
- s_7 = `the man sleeps`. (We have used the rule `Vi` → `sleeps`.)

It is very convenient to represent derivations as *parse trees*. For example, the above derivation would be represented as the parse tree shown in figure 2. This parse tree has `S` as its root, reflecting the fact that s_1 = `S`. We see the sequence `NP VP` directly below `S`, reflecting the fact that the `S` was expanded using the rule `S` → `NP VP`; we see the sequence `DT NN` directly below the `NP`, reflecting the fact that the `NP` was expanded using the rule `NP` → `DT NN`; and so on.

A context-free grammar G will in general specify a set of possible left-most derivations. Each left-most derivation will end in a string $s_n \in \Sigma^*$: we say that s_n

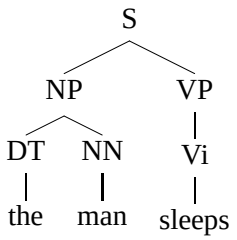


Figure 2: A derivation can be represented as a parse tree.

is the *yield* of the derivation. The set of possible derivations may be a finite or an infinite set (in fact the set of derivations for the grammar in figure 1 is infinite).

The following definition is crucial:

- A string $s \in \Sigma^*$ is said to be in the *language* defined by the CFG, if there is at least one derivation whose yield is s.

2 Ambiguity

Note that some strings s may have more than one underlying derivation (i.e., more than one derivation with s as the yield). In this case we say that the string is *ambiguous* under the CFG.

As one example, see figure 3, which gives two parse trees for the string *the man saw the dog with the telescope*, both of which are valid under the CFG given in figure 1. This example is a case of prepositional phrase attachment ambiguity: the prepositional phrase (`PP`) *with the telescope* can modify either *the dog*, or *saw the dog*. In the first parse tree shown in the figure, the `PP` modifies *the dog*, leading to an `NP` *the dog with the telescope*: this parse tree corresponds to an interpretation where the dog is holding the telescope. In the second parse tree, the `PP` modifies the entire `VP` *saw the dog*: this parse tree corresponds to an interpretation where the man is using the telescope to see the dog.

Ambiguity is an astonishingly severe problem for natural languages. When researchers first started building reasonably large grammars for languages such as English, they were surprised to see that sentences often had a very large number of possible parse trees: it is not uncommon for a moderate-length sentence (say 20 or 30 words in length) to have hundreds, thousands, or even tens of thousands of possible parses.

As one example, in lecture we argued that the following sentence has a surprisingly large number of parse trees (I've found 14 in total):

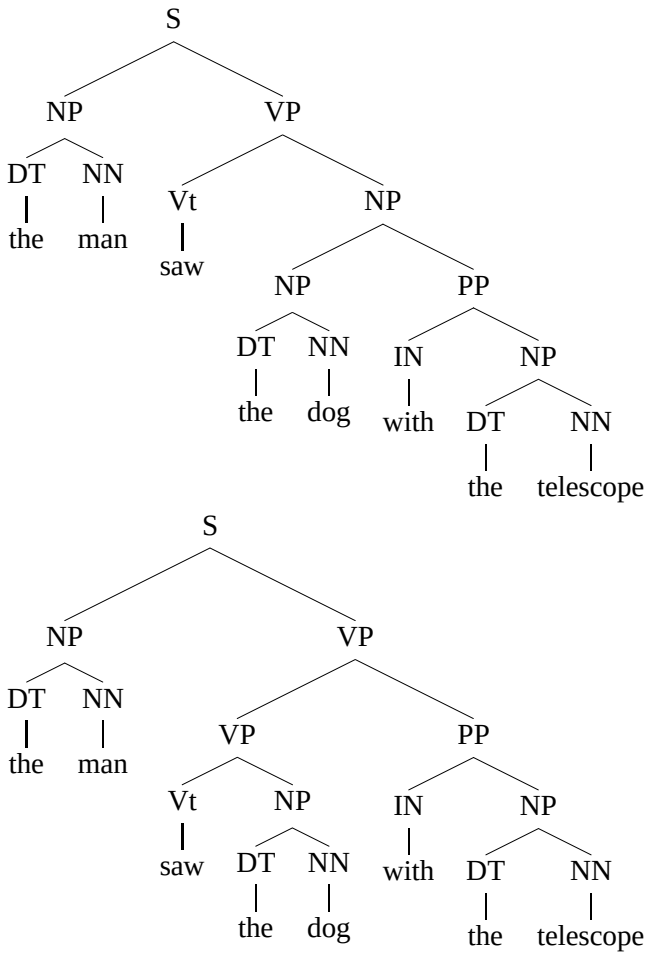


Figure 3: Two parse trees (derivations) for the sentence *the man saw the dog with the telescope*, under the CFG in figure 1.

She announced a program to promote safety in trucks and vans

Can you find the different parse trees for this example?

3 Probabilistic Context-Free Grammars (PCFGs)

3.1 Basic Definitions

Given a context-free grammar G, we will use the following definitions:

- $\mathcal{T}_G$ is the set of all possible left-most derivations (parse trees) under the grammar G. When the grammar G is clear from context we will often write this as simply $\mathcal{T}$.
- For any derivation $t \in \mathcal{T}_G$, we write $\mathsf{yield}(t)$ to denote the string $s \in \Sigma^*$ that is the yield of t (i.e., $\mathsf{yield}(t)$ is the sequence of words in t).
- For a given sentence $s \in \Sigma^*$, we write $\mathcal{T}_G(s)$ to refer to the set

$$\{t : t \in \mathcal{T}_G, \mathsf{yield}(t) = s\}$$

  That is, $\mathcal{T}_G(s)$ is the set of possible parse trees for s.
- We say that a sentence s is *ambiguous* if it has more than one parse tree, i.e., $|\mathcal{T}_G(s)| > 1$.
- We say that a sentence s is *grammatical* if it has at least one parse tree, i.e., $|\mathcal{T}_G(s)| > 0$.

The key idea in probabilistic context-free grammars is to extend our definition to give a *probability distribution over possible derivations*. That is, we will find a way to define a distribution over parse trees, $p(t)$, such that for any $t \in \mathcal{T}_G$,

$$p(t) \geq 0$$

and in addition such that

$$\sum_{t \in \mathcal{T}_G} p(t) = 1$$

At first glance this seems difficult: each parse-tree t is a complex structure, and the set $\mathcal{T}_G$ will most likely be infinite. However, we will see that there is a very simple extension to context-free grammars that allows us to define a function $p(t)$.

Why is this a useful problem? A crucial idea is that once we have a function $p(t)$, we have a ranking over possible parses for any sentence in order of probability. In particular, given a sentence s, we can return

$$\arg \max_{t \in \mathcal{T}_G(s)} p(t)$$

as the output from our parser—this is the most likely parse tree for s under the model. Thus if our distribution $p(t)$ is a good model for the probability of different parse trees in our language, we will have an effective way of dealing with ambiguity.

This leaves us with the following questions:

- How do we define the function $p(t)$?
- How do we learn the parameters of our model of $p(t)$ from training examples?
- For a given sentence s, how do we find the most likely tree, namely

$$\arg \max_{t \in \mathcal{T}_G(s)} p(t)?$$

  This last problem will be referred to as the *decoding* or *parsing* problem.

In the following sections we answer these questions through defining *probabilistic context-free grammars* (PCFGs), a natural generalization of context-free grammars.

3.2 Definition of PCFGs

Probabilistic context-free grammars (PCFGs) are defined as follows:

Definition 1 (PCFGs) *A PCFG consists of:*

1. *A context-free grammar* $G = (N, \Sigma, S, R)$.
2. *A parameter*

   $$q(\alpha \rightarrow \beta)$$

   for each rule $\alpha \rightarrow \beta \in R$. *The parameter* $q(\alpha \rightarrow \beta)$ *can be interpreted as the conditional probabilty of choosing rule* $\alpha \rightarrow \beta$ *in a left-most derivation, given that the non-terminal being expanded is* α. *For any* $X \in N$, *we have the constraint*

   $$\sum_{\alpha \rightarrow \beta \in R : \alpha = X} q(\alpha \rightarrow \beta) = 1$$

   In addition we have $q(\alpha \rightarrow \beta) \geq 0$ *for any* $\alpha \rightarrow \beta \in R$.

Given a parse-tree $t \in \mathcal{T}_G$ containing rules $\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_n \rightarrow \beta_n$, the probability of t under the PCFG is

$$p(t) = \prod_{i=1}^{n} q(\alpha_i \rightarrow \beta_i)$$

□

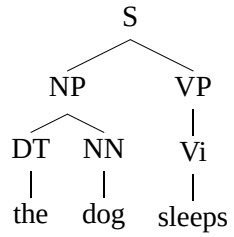
Figure 4 shows an example PCFG, which has the same underlying context-free grammar as that shown in figure 1. The only addition to the original context-free grammar is a parameter $q(\alpha \rightarrow \beta)$ for each rule $\alpha \rightarrow \beta \in R$. Each of these parameters is constrained to be non-negative, and in addition we have the constraint that for any non-terminal $X \in N$,

$$\sum_{\alpha \rightarrow \beta \in R : \alpha = X} q(\alpha \rightarrow \beta) = 1$$

This simply states that for any non-terminal X, the parameter values for all rules with that non-terminal on the left-hand-side of the rule must sum to one. We can verify that this property holds for the PCFG in figure 4. For example, we can verify that this constraint holds for $X = \texttt{VP}$, because

$$\begin{aligned} \sum_{\alpha \rightarrow \beta \in R : \alpha = \texttt{VP}} q(\alpha \rightarrow \beta) &= q(\texttt{VP} \rightarrow \texttt{Vi}) + q(\texttt{VP} \rightarrow \texttt{Vt NP}) + q(\texttt{VP} \rightarrow \texttt{VP PP}) \\ &= 0.3 + 0.5 + 0.2 \\ &= 1.0 \end{aligned}$$

To calculate the probability of any parse tree t, we simply multiply together the q values for the context-free rules that it contains. For example, if our parse tree t is



then we have

$$\begin{aligned} p(t) = \; & q(\texttt{S} \rightarrow \texttt{NP VP}) \times q(\texttt{NP} \rightarrow \texttt{DT NN}) \times q(\texttt{DT} \rightarrow \texttt{the}) \times q(\texttt{NN} \rightarrow \texttt{dog}) \times \\ & q(\texttt{VP} \rightarrow \texttt{Vi}) \times q(\texttt{Vi} \rightarrow \texttt{sleeps}) \end{aligned}$$

Intuitively, PCFGs make the assumption that parse trees are generated stochastically, according to the following process:

N = {S, NP, VP, PP, DT, Vi, Vt, NN, IN}
S = S
Σ = {sleeps, saw, man, woman, dog, telescope, the, with, in}

$R, q =$

S	→	NP	VP	1.0
VP	→	Vi		0.3
VP	→	Vt	NP	0.5
VP	→	VP	PP	0.2
NP	→	DT	NN	0.8
NP	→	NP	PP	0.2
PP	→	IN	NP	1.0

Vi	→	sleeps	1.0
Vt	→	saw	1.0
NN	→	man	0.1
NN	→	woman	0.1
NN	→	telescope	0.3
NN	→	dog	0.5
DT	→	the	1.0
IN	→	with	0.6
IN	→	in	0.4

Figure 4: A simple probabilistic context-free grammar (PCFG). In addition to the set of rules R, we show the parameter value for each rule. For example, $q(\texttt{VP} \rightarrow \texttt{Vt NP}) = 0.5$ in this PCFG.

- Define $s_1 = S$, $i = 1$.
- While s_i contains at least one non-terminal:
  - Find the left-most non-terminal in s_i, call this X.
  - Choose one of the rules of the form $X \rightarrow \beta$ from the distribution $q(X \rightarrow \beta)$.
  - Create s_{i+1} by replacing the left-most X in s_i by β.
  - Set $i = i + 1$.

So we have simply added probabilities to each step in left-most derivations. The probability of an entire tree is the product of probabilities for these individual choices.

3.3 Deriving a PCFG from a Corpus

Having defined PCFGs, the next question is the following: how do we derive a PCFG from a corpus? We will assume a set of training data, which is simply a set

of parse trees $t_1, t_2, \ldots, t_m$. As before, we will write $\mathsf{yield}(t_i)$ to be the yield for the i'th parse tree in the sentence, i.e., $\mathsf{yield}(t_i)$ is the i'th sentence in the corpus.

Each parse tree t_i is a sequence of context-free rules: we assume that every parse tree in our corpus has the same symbol, S, at its root. We can then define a PCFG (N, Σ, S, R, q) as follows:

- N is the set of all non-terminals seen in the trees $t_1 \ldots t_m$.
- Σ is the set of all words seen in the trees $t_1 \ldots t_m$.
- The start symbol S is taken to be S.
- The set of rules R is taken to be the set of all rules $\alpha \rightarrow \beta$ seen in the trees $t_1 \ldots t_m$.
- The maximum-likelihood parameter estimates are

$$q_{ML}(\alpha \rightarrow \beta) = \frac{\text{Count}(\alpha \rightarrow \beta)}{\text{Count}(\alpha)}$$

  where $\text{Count}(\alpha \rightarrow \beta)$ is the number of times that the rule $\alpha \rightarrow \beta$ is seen in the trees $t_1 \ldots t_m$, and $\text{Count}(\alpha)$ is the number of times the non-terminal α is seen in the trees $t_1 \ldots t_m$.

For example, if the rule $\texttt{VP} \rightarrow \texttt{Vt NP}$ is seen 105 times in our corpus, and the non-terminal VP is seen 1000 times, then

$$q(\texttt{VP} \rightarrow \texttt{Vt NP}) = \frac{105}{1000}$$

3.4 Parsing with PCFGs

A crucial question is the following: given a sentence s, how do we find the highest scoring parse tree for s, or more explicitly, how do we find

$$\arg \max_{t \in \mathcal{T}(s)} p(t) \quad ?$$

This section describes a dynamic programming algorithm, *the CKY algorithm*, for this problem.

The CKY algorithm we present applies to a restricted type of PCFG: a PCFG where which is in Chomsky normal form (CNF). While the restriction to grammars in CNF might at first seem to be restrictive, it turns out not to be a strong assumption. It is possible to convert any PCFG into an equivalent grammar in CNF: we will look at this question more in the homeworks.

In the next sections we first describe the idea of grammars in CNF, then describe the CKY algorithm.

N = {S, NP, VP, PP, DT, Vi, Vt, NN, IN}
S = S
Σ = {sleeps, saw, man, woman, dog, telescope, the, with, in}

$R, q =$

S	$\rightarrow$	NP	VP	1.0
VP	$\rightarrow$	Vt	NP	0.8
VP	$\rightarrow$	VP	PP	0.2
NP	$\rightarrow$	DT	NN	0.8
NP	$\rightarrow$	NP	PP	0.2
PP	$\rightarrow$	IN	NP	1.0

Vi	$\rightarrow$	sleeps	1.0
Vt	$\rightarrow$	saw	1.0
NN	$\rightarrow$	man	0.1
NN	$\rightarrow$	woman	0.1
NN	$\rightarrow$	telescope	0.3
NN	$\rightarrow$	dog	0.5
DT	$\rightarrow$	the	1.0
IN	$\rightarrow$	with	0.6
IN	$\rightarrow$	in	0.4

Figure 5: A simple probabilistic context-free grammar (PCFG) in Chomsky normal form. Note that each rule in the grammar takes one of two forms: $X \rightarrow Y_1 \ \ Y_2$ where $X \in N, Y_1 \in N, Y_2 \in N$; or $X \rightarrow Y$ where $X \in N$, $Y \in \Sigma$.

3.4.1 Chomsky Normal Form

Definition 2 (Chomsky Normal Form) *A context-free grammar $G = (N, \Sigma, R, S)$ is in Chomsky form if each rule $\alpha \rightarrow \beta \in R$ takes one of the two following forms:*

- $X \rightarrow Y_1 Y_2$ *where* $X \in N, Y_1 \in N, Y_2 \in N$.
- $X \rightarrow Y$ *where* $X \in N$, $Y \in \Sigma$.

Hence each rule in the grammar either consists of a non-terminal X rewriting as exactly two non-terminal symbols, $Y_1 Y_2$; or a non-terminal X rewriting as exactly one terminal symbol Y. □

Figure 5 shows an example of a PCFG in Chomsky normal form.

3.4.2 Parsing using the CKY Algorithm

We now describe an algorithm for parsing with a PCFG in CNF. The input to the algorithm is a PCFG $G = (N, \Sigma, S, R, q)$ in Chomsky normal form, and a sentence

$s = x_1 \ldots x_n$, where x_i is the i'th word in the sentence. The output of the algorithm is

$$\arg \max_{t \in \mathcal{T}_G(s)} p(t)$$

The CKY algorithm is a dynamic-programming algorithm. Key definitions in the algorithm are as follows:

- For a given sentence $x_1 \ldots x_n$, define $\mathcal{T}(i, j, X)$ for any $X \in N$, for any (i, j) such that $1 \leq i \leq j \leq n$, to be the set of all parse trees for words $x_i \ldots x_j$ such that non-terminal X is at the root of the tree.

- Define

$$\pi(i, j, X) = \max_{t \in \mathcal{T}(i,j,X)} p(t)$$

  (we define $\pi(i, j, X) = 0$ if $\mathcal{T}(i, j, X)$ is the empty set).

Thus $\pi(i, j, X)$ is the highest score for any parse tree that dominates words $x_i \ldots x_j$, and has non-terminal X as its root. The score for a tree t is again taken to be the product of scores for the rules that it contains (i.e. if the tree t contains rules $\alpha_1 \rightarrow \beta_1, \alpha_2 \rightarrow \beta_2, \ldots, \alpha_m \rightarrow \beta_m$, then $p(t) = \prod_{i=1}^{m} q(\alpha_i \rightarrow \beta_i)$).

Note in particular, that

$$\pi(1, n, S) = \arg \max_{t \in \mathcal{T}_G(s)}$$

because by definition $\pi(1, n, S)$ is the score for the highest probability parse tree spanning words $x_1 \ldots x_n$, with S as its root.

The key observation in the CKY algorithm is that we can use a recursive definition of the π values, which allows a simple bottom-up dynamic programming algorithm. The algorithm is "bottom-up", in the sense that it will first fill in $\pi(i, j, X)$ values for the cases where $j = i$, then the cases where $j = i + 1$, and so on.

The base case in the recursive definition is as follows: for all $i = 1 \ldots n$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

This is a natural definition: the only way that we can have a tree rooted in node X spanning word x_i is if the rule $X \rightarrow x_i$ is in the grammar, in which case the tree has score $q(X \rightarrow x_i)$; otherwise, we set $\pi(i, i, X) = 0$, reflecting the fact that there are no trees rooted in X spanning word x_i.

The recursive definition is as follows: for all (i, j) such that $1 \leq i < j \leq n$, for all $X \in N$,

$$\pi(i, j, X) = \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \ldots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z)) \quad (1)$$

The next section of this note gives justification for this recursive definition.

Figure 6 shows the final algorithm, based on these recursive definitions. The algorithm fills in the π values bottom-up: first the $\pi(i, i, X)$ values, using the base case in the recursion; then the values for $\pi(i, j, X)$ such that $j = i + 1$; then the values for $\pi(i, j, X)$ such that $j = i + 2$; and so on.

Note that the algorithm also stores *backpointer* values $bp(i, j, X)$ for all values of (i, j, X). These values record the rule $X \rightarrow YZ$ and the split-point s leading to the highest scoring parse tree. The backpointer values allow recovery of the highest scoring parse tree for the sentence.

3.4.3 Justification for the Algorithm

As an example of how the recursive rule in Eq. 2 is applied, consider parsing the sentence

$$x_1 \ldots x_8 = \textit{the dog saw the man with the telescope}$$

and consider the calculation of $\pi(3, 8, \texttt{VP})$. This will be the highest score for any tree with root VP, spanning words $x_3 \ldots x_8 =$ *saw the man with the telescope*. Eq. 2 specifies that to calculate this value we take the $\max$ over two choices: first, a choice of a rule $\texttt{VP} \rightarrow YZ$ which is in the set of rules R—note that there are two such rules, $\texttt{VP} \rightarrow \texttt{Vt NP}$ and $\texttt{VP} \rightarrow \texttt{VP PP}$. Second, a choice of $s \in \{3, 4, \ldots 7\}$. Thus we will take the maximum value of the following terms:

$$\begin{aligned}
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 3, \texttt{Vt}) \times \pi(4, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 3, \texttt{VP}) \times \pi(4, 8, \texttt{PP}) \\
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 4, \texttt{Vt}) \times \pi(5, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 4, \texttt{VP}) \times \pi(5, 8, \texttt{PP}) \\
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 5, \texttt{Vt}) \times \pi(6, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 5, \texttt{VP}) \times \pi(6, 8, \texttt{PP}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
&q(\texttt{VP} \rightarrow \texttt{Vt NP}) \times \pi(3, 7, \texttt{Vt}) \times \pi(8, 8, \texttt{NP}) \\
&q(\texttt{VP} \rightarrow \texttt{VP PP}) \times \pi(3, 7, \texttt{VP}) \times \pi(8, 8, \texttt{PP})
\end{aligned}$$

Input: a sentence $s = x_1 \dots x_n$, a PCFG $G = (N, \Sigma, S, R, q)$.
Initialization:
For all $i \in \{1 \dots n\}$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

Algorithm:

- For $l = 1 \dots (n-1)$
  - For $i = 1 \dots (n-l)$
    - Set $j = i + l$
    - For all $X \in N$, calculate

$$\pi(i, j, X) = \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

and

$$bp(i, j, X) = \arg \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

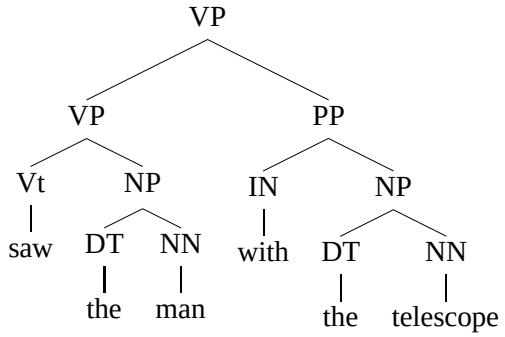
Output: Return $\pi(1, n, S) = \max_{t \in \mathcal{T}(s)} p(t)$, and backpointers bp which allow recovery of $\arg\max_{t \in \mathcal{T}(s)} p(t)$.

Figure 6: The CKY parsing algorithm.

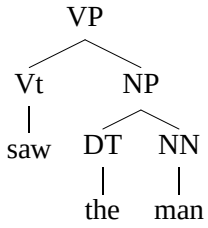
How do we justify this recursive definition? The key observation is that any tree t rooted in X, spanning words $x_i \ldots x_j$, must consist of the following:

- A choice of some rule $X \rightarrow Y\ Z \in R$, at the top of the tree.
- A choice of some value $s \in \{i \ldots j-1\}$, which we will refer to as the "split point" of the rule.
- A choice of a tree rooted in Y, spanning words $x_i \ldots x_s$, call this tree t_1
- A choice of a tree rooted in Z, spanning words $x_{s+1} \ldots x_j$, call this tree t_2.
- We then have
$$p(t) = q(X \rightarrow Y\ Z) \times p(t_1) \times p(t_2)$$
I.e., the probability for the tree t is the product of three terms: the rule probability for the rule at the top of the tree, and probabilities for the sub-trees t_1 and t_2.

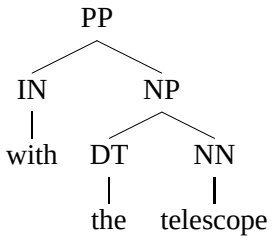
For example, consider the following tree, rooted in `VP`, spanning words $x_3 \ldots x_8$ in our previous example:



In this case we have the rule `VP` $\rightarrow$ `VP PP` at the top of the tree; the choice of split-point is $s = 5$; the tree dominating words $x_3 \ldots x_s$, rooted in `VP`, is



and the tree dominating words $x_{s+1} \ldots x_8$, rooted in `PP`, is



The second key observation is the following:

- *If the highest scoring tree rooted in non-terminal X, and spanning words $x_i \dots x_j$, uses rule $X \rightarrow Y \ \ Z$ and split point s, then its two subtrees must be: 1) the highest scoring tree rooted in Y that spanns words $x_i \dots x_s$; 2) the highest scoring tree rooted in Z that spans words $x_{s+1} \dots x_j$.*

The proof is by contradiction. If either condition (1) or condition (2) was not true, we could always find a higher scoring tree rooted in X, spanning words $x_i \dots x_j$, by choosing a higher scoring subtree spanning words $x_i \dots x_s$ or $x_{s+1} \dots x_j$.

Now let's look back at our recursive definition:

$$\pi(i, j, X) = \max_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

We see that it involves a search over rules possible rules $X \rightarrow YZ \in R$, and possible split points s. For each choice of rule and split point, we calculate

$$q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z)$$

which is the highest scoring tree rooted in X, spanning words $x_i \dots x_j$, with this choice of rule and split point. The definition uses the values $\pi(i, s, Y)$ and $\pi(s+1, j, Z)$, corresponding to the two highest scoring subtrees. We take the $\max$ over all possible choices of rules and split points.

3.4.4 The Inside Algorithm for Summing over Trees

We now describe a second, very similar algorithm, which sums the probabilities for all parse trees for a given sentence, thereby calculating the probability of the sentence under the PCFG. The algorithm is called *the inside algorithm*.

The input to the algorithm is again a PCFG $G = (N, \Sigma, S, R, q)$ in Chomsky normal form, and a sentence $s = x_1 \dots x_n$, where x_i is the i'th word in the sentence. The output of the algorithm is

$$p(s) = \sum_{t \in \mathcal{T}_G(s)} p(t)$$

Here $p(s)$ is the probability of the PCFG generating string s.

We define the following:

- As before, for a given sentence $x_1 \ldots x_n$, define $\mathcal{T}(i, j, X)$ for any $X \in N$, for any (i, j) such that $1 \leq i \leq j \leq n$, to be the set of all parse trees for words $x_i \ldots x_j$ such that non-terminal X is at the root of the tree.

- Define
$$\pi(i, j, X) = \sum_{t \in \mathcal{T}(i,j,X)} p(t)$$
(we define $\pi(i, j, X) = 0$ if $\mathcal{T}(i, j, X)$ is the empty set).

Note that we have simply replaced the $\max$ in the previous definition of π, with a sum.

In particular, we have

$$\pi(1, n, S) = \sum_{t \in \mathcal{T}_G(s)} p(t) = p(s)$$

Thus by calculating $\pi(1, n, S)$, we have calculated the probability $p(s)$.

We use a very similar recursive definition to before. First, the base case is as follows: for all $i = 1 \ldots n$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

The recursive definition is as follows: for all (i, j) such that $1 \leq i < j \leq n$, for all $X \in N$,

$$\pi(i, j, X) = \sum_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \ldots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s + 1, j, Z)) \quad (2)$$

Figure 7 shows the algorithm based on these recursive definitions. The algorithm is essentially identical to the CKY algorithm, but with $\max$ replaced by a sum in the recursive definition. The π values are again calculated bottom-up.

Input: a sentence $s = x_1 \dots x_n$, a PCFG $G = (N, \Sigma, S, R, q)$.
Initialization:
For all $i \in \{1 \dots n\}$, for all $X \in N$,

$$\pi(i, i, X) = \begin{cases} q(X \rightarrow x_i) & \text{if } X \rightarrow x_i \in R \\ 0 & \text{otherwise} \end{cases}$$

Algorithm:

- For $l = 1 \dots (n-1)$
  - For $i = 1 \dots (n-l)$
    - Set $j = i + l$
    - For all $X \in N$, calculate

$$\pi(i, j, X) = \sum_{\substack{X \rightarrow YZ \in R, \\ s \in \{i \dots (j-1)\}}} (q(X \rightarrow YZ) \times \pi(i, s, Y) \times \pi(s+1, j, Z))$$

Output: Return $\pi(1, n, S) = \sum_{t \in \mathcal{T}(s)} p(t)$

Figure 7: The inside algorithm.

The Naive Bayes Model, Maximum-Likelihood Estimation, and the EM Algorithm

Michael Collins

1 Introduction

This note covers the following topics:

- The Naive Bayes model for classification (with text classification as a specific example).
- The derivation of maximum-likelihood (ML) estimates for the Naive Bayes model, in the simple case where the underlying labels are observed in the training data.
- The EM algorithm for parameter estimation in Naive Bayes models, in the case where labels are missing from the training examples.
- The EM algorithm in general form, including a derivation of some of its convergence properties.

We will use the Naive Bayes model throughout this note, as a simple model where we can derive the EM algorithm. Later in the class we will consider EM for parameter estimation of more complex models, for example hidden Markov models and probabilistic context-free grammars.

2 The Naive Bayes Model for Classification

This section describes a model for binary classification, *Naive Bayes*. Naive Bayes is a simple but important probabilistic model. It will be used as a running example in this note. In particular, we will first consider maximum-likelihood estimation in the case where the data is "fully observed"; we will then consider the expectation maximization (EM) algorithm for the case where the data is "partially observed", in the sense that the labels for examples are missing.

The setting is as follows. Assume we have some training set $(\underline{x}^{(i)}, y^{(i)})$ for $i = 1 \ldots n$, where each $\underline{x}^{(i)}$ is a vector, and each $y^{(i)}$ is in $\{1, 2, \ldots, k\}$. Here k is an integer specifying the number of classes in the problem. This is a *multiclass* classification problem, where the task is to map each input vector $\underline{x}$ to a label y that can take any one of k possible values. (For the special case of $k = 2$ we have a binary classification problem.)

We will assume throughout that each vector $\underline{x}$ is in the set $\{-1, +1\}^d$ for some integer d specifying the number of "features" in the model. In other words, each component x_j for $j = 1 \ldots d$ can take one of two possible values.

As one example motivating this setting, consider the problem of classifying documents into k different categories (for example $y = 1$ might correspond to a *sports* category, $y = 2$ might correspond to a *music* category, $y = 3$ might correspond to a *current affairs* category, and so on). The label $y^{(i)}$ represents the category of the i'th document in the collection. Each component $x_j^{(i)}$ for $j = 1 \ldots d$ might represent the presence or absence of a particular word. For example we might define $x_1^{(i)}$ to be $+1$ if the i'th document contains the word *Giants*, or -1 otherwise; $x_2^{(i)}$ to be $+1$ if the i'th document contains the word *Obama*, or -1 otherwise; and so on.

The Naive Bayes model is then derived as follows. We assume random variables Y and $X_1 \ldots X_d$ corresponding to the label y and the vector components $x_1, x_2, \ldots, x_d$. Our task will be to model the joint probability

$$P(Y = y, X_1 = x_1, X_2 = x_2, \ldots X_d = x_d)$$

for any label y paired with attribute values $x_1 \ldots x_d$. A key idea in the NB model is the following assumption:

$$\begin{aligned} & P(Y = y, X_1 = x_1, X_2 = x_2, \ldots X_d = x_d) \\ = \quad & P(Y = y) \prod_{j=1}^{d} P(X_j = x_j | Y = y) \end{aligned} \tag{1}$$

This equality is derived using independence assumptions, as follows. First, by the chain rule, the following identity is exact (any joint distribution over $Y, X_1 \ldots X_d$ can be factored in this way):

$$\begin{aligned} & P(Y = y, X_1 = x_1, X_2 = x_2, \ldots X_d = x_d) \\ = \quad & P(Y = y) \times P(X_1 = x_1, X_2 = x_2, \ldots X_d = x_d | Y = y) \end{aligned}$$

Next, we deal with the second term as follows:

$$P(X_1 = x_1, X_2 = x_2, \ldots X_d = x_d | Y = y)$$

$$= \prod_{j=1}^{d} P(X_j = x_j | X_1 = x_1, X_2 = x_2, \ldots X_{j-1} = x_{j-1}, Y = y)$$
$$= \prod_{j=1}^{d} P(X_j = x_j | Y = y)$$

The first equality is again exact, by the chain rule. The second equality follows by an independence assumption, namely that for all $j = 1 \ldots d$, the value for the random variable X_j is independent of all other attribute values, $X_{j'}$ for $j' \neq j$, when conditioned on the identity of the label Y. This is the *Naive Bayes* assumption. It is naive, in the sense that it is a relatively strong assumption. It is, however, a very useful assumption, in that it dramatically reduces the number of parameters in the model, while still leading to a model that can be quite effective in practice.

Following Eq. 1, the NB model has two types of parameters: $q(y)$ for $y \in \{1 \ldots k\}$, with

$$P(Y = y) = q(y)$$

and $q_j(x|y)$ for $j \in \{1 \ldots d\}$, $x \in \{-1, +1\}$, $y \in \{1 \ldots k\}$, with

$$P(X_j = x | Y = y) = q_j(x|y)$$

We then have

$$p(y, x_1 \ldots x_d) = q(y) \prod_{j=1}^{d} q_j(x_j|y)$$

To summarize, we give the following definition:

Definition 1 (Naive Bayes (NB) Model) *A NB model consists of an integer k specifying the number of possible labels, an integer d specifying the number of attributes, and in addition the following parameters:*

- *A parameter*
$$q(y)$$
for any $y \in \{1 \ldots k\}$. The parameter $q(y)$ can be interpreted as the probability of seeing the label y. We have the constraints $q(y) \geq 0$ and $\sum_{y=1}^{k} q(y) = 1$.

- *A parameter*
$$q_j(x|y)$$
for any $j \in \{1 \ldots d\}$, $x \in \{-1, +1\}$, $y \in \{1 \ldots k\}$. The value for $q_j(x|y)$ can be interpreted as the probability of attribute j taking value x, conditioned on the underlying label being y. We have the constraints that $q_j(x|y) \geq 0$, and for all y, j, $\sum_{x \in \{-1,+1\}} q_j(x|y) = 1$.

We then define the probability for any $y, x_1 \ldots x_d$ as

$$p(y, x_1 \ldots x_d) = q(y) \prod_{j=1}^{d} q_j(x_j|y) \qquad \square$$

The next section describes how the parameters can be estimated from training examples. Once the parameters have been estimated, given a new test example $\underline{x} = \langle x_1, x_2, \ldots, x_d \rangle$, the output of the NB classifier is

$$\arg \max_{y \in \{1 \ldots k\}} p(y, x_1 \ldots x_d) = \arg \max_{y \in \{1 \ldots k\}} \left(q(y) \prod_{j=1}^{d} q_j(x_j|y) \right)$$

3 Maximum-Likelihood estimates for the Naive Bayes Model

We now consider how the parameters $q(y)$ and $q_j(x|y)$ can be estimated from data. In particular, we will describe the *maximum-likelihood estimates*. We first state the form of the estimates, and then go into some detail about how the estimates are derived.

Our training sample consists of examples $(\underline{x}^{(i)}, y^{(i)})$ for $i = 1 \ldots n$. Recall that each $\underline{x}^{(i)}$ is a d-dimensional vector. We write $x_j^{(i)}$ for the value of the j'th component of $\underline{x}^{(i)}$; $x_j^{(i)}$ can take values -1 or $+1$.

Given these definitions, the maximum-likelihood estimates for $q(y)$ for $y \in \{1 \ldots k\}$ take the following form:

$$q(y) = \frac{\sum_{i=1}^{n} [[y^{(i)} = y]]}{n} = \frac{\text{count}(y)}{n} \tag{2}$$

Here we define $[[y^{(i)} = y]]$ to be 1 if $y^{(i)} = y$, 0 otherwise. Hence $\sum_{i=1}^{n} [[y^{(i)} = y]] = \text{count}(y)$ is simply the number of times that the label y is seen in the training set.

Similarly, the ML estimates for the $q_j(x|y)$ parameters (for all $y \in \{1 \ldots k\}$, for all $x \in \{-1, +1\}$, for all $j \in \{1 \ldots d\}$) take the following form:

$$q_j(x|y) = \frac{\sum_{i=1}^{n} [[y^{(i)} = y \text{ and } x_j^{(i)} = x]]}{\sum_{i=1}^{n} [[y^{(i)} = y]]} = \frac{\text{count}_j(x|y)}{\text{count}(y)} \tag{3}$$

where

$$\text{count}_j(x|y) = \sum_{i=1}^{n} [[y^{(i)} = y \text{ and } x_j^{(i)} = x]]$$

This is a very natural estimate: we simply count the number of times label y is seen in conjunction with x_j taking value x; count the number of times the label y is seen in total; then take the ratio of these two terms.

4 Deriving the Maximum-Likelihood Estimates

4.1 Definition of the ML Estimation Problem

We now describe how the ML estimates in Eqs. 2 and 3 are derived. Given the training set $(x^{(i)}, y^{(i)})$ for $i = 1 \ldots n$, the log-likelihood function is

$$\begin{aligned} L(\underline{\theta}) &= \sum_{i=1}^{n} \log p(x^{(i)}, y^{(i)}) \\ &= \sum_{i=1}^{n} \log \left(q(y^{(i)}) \prod_{j=1}^{d} q_j(x_j^{(i)} | y^{(i)}) \right) \\ &= \sum_{i=1}^{n} \log q(y^{(i)}) + \sum_{i=1}^{n} \log \left(\prod_{j=1}^{d} q_j(x_j^{(i)} | y^{(i)}) \right) \\ &= \sum_{i=1}^{n} \log q(y^{(i)}) + \sum_{i=1}^{n} \sum_{j=1}^{d} \log q_j(x_j^{(i)} | y^{(i)}) \end{aligned} \tag{4}$$

Here for convenience we use $\underline{\theta}$ to refer to a parameter vector consisting of values for all parameters $q(y)$ and $q_j(x|y)$ in the model. The log-likelihood is a function of the parameter values, and the training examples. The final two equalities follow from the usual property that $\log(a \times b) = \log a + \log b$.

As usual, the log-likelihood function $L(\underline{\theta})$ can be interpreted as a measure of how well the parameter values fit the training example. In ML estimation we seek the parameter values that maximize $L(\underline{\theta})$.

The maximum-likelihood problem is the following:

Definition 2 (ML Estimates for Naive Bayes Models) *Assume a training set $(x^{(i)}, y^{(i)})$ for $i \in \{1 \ldots n\}$. The maximum-likelihood estimates are then the parameter values $q(y)$ for $y \in \{1 \ldots k\}$, $q_j(x|y)$ for $j \in \{1 \ldots d\}$, $y \in \{1 \ldots k\}$, $x \in \{-1, +1\}$ that maximize*

$$L(\underline{\theta}) = \sum_{i=1}^{n} \log q(y^{(i)}) + \sum_{i=1}^{n} \sum_{j=1}^{d} \log q_j(x_j^{(i)} | y^{(i)})$$

subject to the following constraints:

1. $q(y) \geq 0$ *for all* $y \in \{1 \ldots k\}$. $\sum_{y=1}^{k} q(y) = 1$.
2. *For all* y, j, x, $q_j(x|y) \geq 0$. *For all* $y \in \{1 \ldots k\}$, *for all* $j \in \{1 \ldots d\}$,

$$\sum_{x \in \{-1,+1\}} q_j(x|y) = 1$$

□

A crucial result is then the following:

Theorem 1 *The ML estimates for Naive Bayes models (see definition 2) take the form*

$$q(y) = \frac{\sum_{i=1}^{n}[[y^{(i)} = y]]}{n} = \frac{\mathit{count}(y)}{n}$$

and

$$q_j(x|y) = \frac{\sum_{i=1}^{n}[[y^{(i)} = y \text{ and } x_j^{(i)} = x]]}{\sum_{i=1}^{n}[[y^{(i)} = y]]} = \frac{\mathit{count}_j(x|y)}{\mathit{count}(y)}$$

I.e., they take the form given in Eqs. 2 and 3.

The remainder of this section proves the result in theorem 1. We first consider a simple but crucial result, concerning ML estimation for multinomial distributions. We then see how this result leads directly to a proof of theorem 1.

4.2 Maximum-likelihood Estimation for Multinomial Distributions

Consider the following setting. We have some finite set $\mathcal{Y}$. A *distribution* over the set $\mathcal{Y}$ is a vector q with components q_y for each $y \in \mathcal{Y}$, corresponding to the probability of seeing element y. We define $\mathcal{P}_\mathcal{Y}$ to be the set of all distributions over the set $\mathcal{Y}$: that is,

$$\mathcal{P}_\mathcal{Y} = \{q \in \mathbb{R}^{|\mathcal{Y}|} : \forall y \in \mathcal{Y}, q_y \geq 0; \ \sum_{y \in \mathcal{Y}} q_y = 1\}$$

In addition, assume that we have some vector c with components c_y for each $y \in \mathcal{Y}$. We will assume that each $c_y \geq 0$. In many cases c_y will correspond to some "count" taken from data: specifically the number of times that we see element y. We also assume that there is at least one $y \in \mathcal{P}_\mathcal{Y}$ such that $c_y > 0$ (i.e., such that c_y is strictly positive).

We then state the following optimization problem:

Definition 3 (ML estimation problem for multinomials) *The input to the problem is a finite set $\mathcal{Y}$, and a weight $c_y \geq 0$ for each $y \in \mathcal{Y}$. The output from the problem is the distribution q^* that solves the following maximization problem:*

$$q^* = \arg\max_{q \in \mathcal{P}_\mathcal{Y}} \sum_{y \in \mathcal{Y}} c_y \log q_y$$

Thus the optimal vector q^* is a distribution (it is a member of the set $\mathcal{P}_\mathcal{Y}$), and in addition it maximizes the function $\sum_{y\in\mathcal{Y}} c_y \log q_y$.

We give a theorem that gives a very simple (and intuitive) form for q^*:

Theorem 2 *Consider the problem in definition 3. The vector q^* has components*

$$q_y^* = \frac{c_y}{N}$$

for all $y \in \mathcal{Y}$, where $N = \sum_{y\in\mathcal{Y}} c_y$.

Proof: To recap, our goal is to maximize the function

$$\sum_{y\in\mathcal{Y}} c_y \log q_y$$

subject to the constraints $q_y \geq 0$ and $\sum_{y\in\mathcal{Y}} q_y = 1$. For simplicity we will assume throughout that $c_y > 0$ for all y.[1]

We will introduce a single Lagrange multiplier $\lambda \in \mathbb{R}$ corresponding to the constraint that $\sum_{y\in\mathcal{Y}} q_y = 1$. The Lagrangian is then

$$g(\lambda, q) = \sum_{y\in\mathcal{Y}} c_y \log q_y - \lambda \left(\sum_{y\in\mathcal{Y}} q_y - 1 \right)$$

By the usual theory of Lagrange multipliers, the solution q_y^* to the maximization problem must satisfy the conditions

$$\frac{d}{dq_y} g(\lambda, q) = 0$$

for all y, and

$$\sum_{y\in\mathcal{Y}} q_y = 1 \tag{5}$$

Differentiating with respect to q_y gives

$$\frac{d}{dq_y} g(\lambda, q) = \frac{c_y}{q_y} - \lambda$$

Setting this derivative to zero gives

$$q_y = \frac{c_y}{\lambda} \tag{6}$$

[1] In a full proof it can be shown that for any y such that $c_y = 0$, we must have $q_y = 0$; we can then consider the problem of maximizing $\sum_{y\in\mathcal{Y}'} c_y \log q_y$ subject to $\sum_{y\in\mathcal{Y}'} q_y = 1$, where $\mathcal{Y}' = \{y \in \mathcal{Y} : c_y > 0\}$.

Combining Eqs. 6 and 5 gives

$$q_y = \frac{c_y}{\sum_{y \in \mathcal{Y}} c_y}$$

□

The proof of theorem 1 follows directly from this result. See section A for a full proof.

5 The EM Algorithm for Naive Bayes

Now consider the following setting. We have a training set consisting of vectors $x^{(i)}$ for $i = 1 \ldots n$. As before, each $x^{(i)}$ is a vector with components $x_j^{(i)}$ for $j \in \{1 \ldots d\}$, where each component can take either the value -1 or $+1$. In other words, our training set *does not have any labels*. Can we still estimate the parameters of the model?

As a concrete example, consider a very simple text classification problem where the vector $\underline{x}$ representing a document has the following four components (i.e., $d = 4$):

$$\begin{aligned}
x_1 &= +1 \text{ if the document contains the word } \textit{Obama}, -1 \text{ otherwise} \\
x_2 &= +1 \text{ if the document contains the word } \textit{McCain}, -1 \text{ otherwise} \\
x_3 &= +1 \text{ if the document contains the word } \textit{Giants}, -1 \text{ otherwise} \\
x_4 &= +1 \text{ if the document contains the word } \textit{Patriots}, -1 \text{ otherwise}
\end{aligned}$$

In addition, we assume that our training data consists of the following examples:

$$\begin{aligned}
\underline{x}^{(1)} &= \langle +1, +1, -1, -1 \rangle \\
\underline{x}^{(2)} &= \langle -1, -1, +1, +1 \rangle \\
\underline{x}^{(3)} &= \langle +1, +1, -1, -1 \rangle \\
\underline{x}^{(4)} &= \langle -1, -1, +1, +1 \rangle \\
\underline{x}^{(5)} &= \langle -1, -1, +1, +1 \rangle
\end{aligned}$$

Intuitively, this data might arise because documents 1 and 3 are about politics (and thus include words like *Obama* or *McCain*, which refer to politicians), and documents $2, 4$ and 5 are about sports (and thus include words like *Giants*, or *Patriots*, which refer to sports teams).

For this data, a good setting of the parameters of a NB model might be as follows (we will soon formalize exactly what it means for the parameter values to be a "good" fit to the data):

$$q(1) = \frac{2}{5}; \quad q(2) = \frac{3}{5}; \tag{7}$$

$$q_1(+1|1) = 1; \quad q_2(+1|1) = 1; \quad q_3(+1|1) = 0; \quad q_4(+1|1) = 0; \tag{8}$$

$$q_1(+1|2) = 0; \quad q_2(+1|2) = 0; \quad q_3(+1|2) = 1; \quad q_4(+1|2) = 1 \tag{9}$$

Thus there are two classes of documents. There is a probability of $2/5$ of seeing class 1, versus a probability of $3/5$ of seeing class 2. Given class 1, we have the vector $\underline{x} = \langle +1, +1, -1, -1 \rangle$ with probability 1; conversely, given class 2, we have the vector $\underline{x} = \langle -1, -1, +1, +1 \rangle$ with probability 1.

Remark. Note that an equally good fit to the data would be the parameter values

$$q(2) = \frac{2}{5}; \quad q(1) = \frac{3}{5};$$

$$q_1(+1|2) = 1; \quad q_2(+1|2) = 1; \quad q_3(+1|2) = 0; \quad q_4(+1|2) = 0;$$

$$q_1(+1|1) = 0; \quad q_2(+1|1) = 0; \quad q_3(+1|1) = 1; \quad q_4(+1|1) = 1$$

Here we have just switched the meaning of classes 1 and 2, and permuted all of the associated probabilities. Cases like this, where symmetries mean that multiple models give the same fit to the data, are common in the EM setting.

5.1 The Maximum-Likelihood Problem for Naive Bayes with Missing Labels

We now describe the parameter estimation method for Naive Bayes when the labels $y^{(i)}$ for $i \in \{1 \ldots n\}$ are missing. The first key insight is that for any example $\underline{x}$, the probability of that example under a NB model can be calculated by marginalizing out the labels:

$$p(\underline{x}) = \sum_{y=1}^{k} p(\underline{x}, y) = \sum_{y=1}^{k} \left(q(y) \prod_{j=1}^{d} q_j(x_j|y) \right)$$

Given this observation, we can define a log-likelihood function as follows. The log-likelihood function is again a measure of how well the parameter values fit the

training examples. Given the training set $(x^{(i)})$ for $i = 1 \ldots n$, the log-likelihood function (we again use $\underline{\theta}$ to refer to the full set of parameters in the model) is

$$\begin{aligned} L(\underline{\theta}) &= \sum_{i=1}^{n} \log p(x^{(i)}) \\ &= \sum_{i=1}^{n} \log \sum_{y=1}^{k} \left(q(y) \prod_{j=1}^{d} q_j(x_j^{(i)}|y) \right) \end{aligned}$$

In ML estimation we seek the parameter values that maximize $L(\underline{\theta})$. This leads to the following problem definition:

Definition 4 (ML Estimates for Naive Bayes Models with Missing Labels) *Assume a training set $(x^{(i)})$ for $i \in \{1 \ldots n\}$. The maximum-likelihood estimates are then the parameter values $q(y)$ for $y \in \{1 \ldots k\}$, $q_j(x|y)$ for $j\{1 \ldots d\}$, $y \in \{1 \ldots k\}$, $x \in \{-1, +1\}$ that maximize*

$$L(\underline{\theta}) = \sum_{i=1}^{n} \log \sum_{y=1}^{k} \left(q(y) \prod_{j=1}^{d} q_j(x_j^{(i)}|y) \right) \tag{10}$$

subject to the following constraints:

1. $q(y) \geq 0$ *for all* $y \in \{1 \ldots k\}$. $\sum_{y=1}^{k} q(y) = 1$.
2. *For all* y, j, x, $q_j(x|y) \geq 0$. *For all* $y \in \{1 \ldots k\}$, *for all* $j \in \{1 \ldots d\}$,

$$\sum_{x \in \{-1,+1\}} q_j(x|y) = 1$$

□

Given this problem definition, we are left with the following questions:

How are the ML estimates justified? In a formal sense, the following result holds. Assume that the training examples $x^{(i)}$ for $i = 1 \ldots n$ are actually i.i.d. samples from a distribution specified by a Naive Bayes model. Equivalently, we assume that the training samples are drawn from a process that first generates an $(\underline{x}, y)$ pair, then deletes the value of the label y, leaving us with only the observations $\underline{x}$. Then it can be shown that the ML estimates are *consistent*, in that as the

number of training samples n increases, the parameters will converge to the true values of the underlying Naive Bayes model.[2]

From a more practical point of view, in practice the ML estimates will often uncover useful patterns in the training examples. For example, it can be verified that the parameter values in Eqs. 7, 8 and 9 do indeed maximize the log-likelihood function given the documents given in the example.

How are the ML estimates useful? Assuming that we have an algorithm that calculates the maximum-likelihood estimates, how are these estimates useful? In practice, there are several scenarios in which the maximum-likelihood estimates will be useful. The parameter estimates find useful patterns in the data: for example, in the context of text classification they can find a useful partition of documents in naturally occurring classes. In particular, once the parameters have been estimated, for any document $\underline{x}$, for any class $y \in \{1 \dots k\}$, we can calculate the conditional probability

$$p(y|\underline{x}) = \frac{p(\underline{x}, y)}{\sum_{y=1}^{k} p(\underline{x}, y)}$$

under the model. This allows us to calculate the probability of document $\underline{x}$ falling into cluster y: if we required a hard partition of the documents into k different classes, we could take the highest probability label,

$$\arg \max_{y \in \{1 \dots k\}} p(y|\underline{x})$$

There are many other uses of EM, which we will see later in the course.

Given a training set, how can we calculate the ML estimates? The final question concerns calculation of the ML estimates. To recap, the function that we would like to maximize (see Eq. 10) is

$$L(\underline{\theta}) = \sum_{i=1}^{n} \log \sum_{y=1}^{k} \left(q(y) \prod_{j=1}^{d} q_j(x_j^{(i)}|y) \right)$$

note the contrast with the regular ML problem (see Eq. 4), where we have labels $y^{(i)}$, and the function we wish to optimize is

$$L(\underline{\theta}) = \sum_{i=1}^{n} \log \left(q(y^{(i)}) \prod_{j=1}^{d} q_j(x_j^{(i)}|y^{(i)}) \right)$$

[2] Up to symmetries in the model; for example, in the text classification example given earlier with *Obama, McCain, Giants, Patriots*, either of the two parameter settings given would be recovered.

The two functions are similar, but crucially *the new definition of $L(\underline{\theta})$ has an additional sum over $y = 1 \ldots k$*, which appears within the $\log$. This sum makes optimization of $L(\underline{\theta})$ hard (in contrast to the definition when the labels are observed).

The next section describes the expectation-maximization (EM) algorithm for calculation of the ML estimates. Because of the difficulty of optimizing the new definition of $L(\underline{\theta})$, the algorithm will have relatively weak guarantees, in the sense that it will only be guaranteed to reach a *local optimum* of the function $L(\underline{\theta})$. The EM algorithm is, however, widely used, and can be very effective in practice.

5.2 The EM Algorithm for Naive Bayes Models

The EM algorithm for Naive Bayes models is shown in figure 1. It is an iterative algorithm, defining a series of parameter values $\underline{\theta}^0, \underline{\theta}^1, \ldots, \underline{\theta}^T$. The initial parameter values $\underline{\theta}^0$ are chosen to be random. At each iteration the new parameter values $\underline{\theta}^t$ are calculated as a function of the training set, and the previous parameter values $\underline{\theta}^{t-1}$. A first key step at each iteration is to calculate the values

$$\delta(y|i) = p(y|\underline{x}^{(i)}; \underline{\theta}^{t-1})$$

for each example $i \in \{1 \ldots n\}$, for each possible label $y \in \{1 \ldots k\}$. The value for $\delta(y|i)$ is the conditional probability for label y on the i'th example, given the parameter values $\underline{\theta}^{t-1}$. The second step at each iteration is to calculate the new parameter values, as

$$q^t(y) = \frac{1}{n} \sum_{i=1}^{n} \delta(y|i)$$

and

$$q_j^t(x|y) = \frac{\sum_{i:x_j^i = x} \delta(y|i)}{\sum_i \delta(y|i)}$$

Note that these updates are very similar in form to the ML parameter estimates in the case of fully observed data. In fact, if we have labeled examples $(x^{(i)}, y^{(i)})$ for $i \in \{1 \ldots n\}$, and define

$$\delta(y|i) = 1 \text{ if } y_i = y, 0 \text{ otherwise}$$

then it is easily verified that the estimates would be identical to those given in Eqs. 2 and 3. Thus the new algorithm can be interpreted as a method where we replaced the definition

$$\delta(y|i) = 1 \text{ if } y_i = y, 0 \text{ otherwise}$$

used for labeled data with the definition

$$\delta(y|i) = p(y|\underline{x}^{(i)}; \underline{\theta}^{t-1})$$

for unlabeled data. Thus we have essentially "hallucinated" the $\delta(y|i)$ values, based on the previous parameters, given that we do not have the actual labels $y^{(i)}$.

The next section describes why this method is justified. First, however, we need the following property of the algorithm:

Theorem 3 *The parameter estimates $q^t(y)$ and $q^t(x|y)$ for $t = 1 \ldots T$ are*

$$\underline{\theta}^t = \arg\max_{\underline{\theta}} Q(\underline{\theta}, \underline{\theta}^{t-1})$$

under the constraints $q(y) \geq 0$, $\sum_{y=1}^{k} q(y) = 1$, $q_j(x|y) \geq 0$, $\sum_{x \in \{-1,+1\}} q_j(x|y) = 1$, where

$$\begin{aligned} Q(\underline{\theta}, \underline{\theta}^{t-1}) &= \sum_{i=1}^{n} \sum_{y=1}^{k} p(y|\underline{x}^{(i)}; \underline{\theta}^{t-1}) \log p(x^{(i)}, y; \underline{\theta}) \\ &= \sum_{i=1}^{n} \sum_{y=1}^{k} p(y|\underline{x}^{(i)}; \underline{\theta}^{t-1}) \log \left(q(y) \prod_{j=1}^{d} q_j(x_j^{(i)}|y) \right) \end{aligned}$$

6 The EM Algorithm in General Form

In this section we describe a general form of the EM algorithm; the EM algorithm for Naive Bayes is a special case of this general form. We then discuss convergence properties of the general form, which in turn give convergence guarantees for the EM algorithm for Naive Bayes.

6.1 The Algorithm

The general form of the EM algorithm is shown in figure 2. We assume the following setting:

- We have sets $\mathcal{X}$ and $\mathcal{Y}$, where $\mathcal{Y}$ is a finite set (e.g., $\mathcal{Y} = \{1, 2, \ldots k\}$ for some integer k). We have a *model* $p(x, y; \underline{\theta})$ that assigns a probability to each (x, y) such that $x \in \mathcal{X}$, $y \in \mathcal{Y}$, under parameters $\underline{\theta}$. Here we use $\underline{\theta}$ to refer to a vector including all parameters in the model.

Inputs: An integer k specifying the number of classes. Training examples $(x^{(i)})$ for $i = 1 \ldots n$ where each $x^{(i)} \in \{-1, +1\}^d$. A parameter T specifying the number of iterations of the algorithm.

Initialization: Set $q^0(y)$ and $q_j^0(x|y)$ to some initial values (e.g., random values) satisfying the constraints

- $q^0(y) \geq 0$ for all $y \in \{1 \ldots k\}$. $\sum_{y=1}^{k} q^0(y) = 1$.
- For all y, j, x, $q_j^0(x|y) \geq 0$. For all $y \in \{1 \ldots k\}$, for all $j \in \{1 \ldots d\}$,

$$\sum_{x \in \{-1,+1\}} q_j^0(x|y) = 1$$

Algorithm:
For $t = 1 \ldots T$

1. For $i = 1 \ldots n$, for $y = 1 \ldots k$, calculate

$$\delta(y|i) = p(y|\underline{x}^{(i)}; \underline{\theta}^{t-1}) = \frac{q^{t-1}(y) \prod_{j=1}^{d} q_j^{t-1}(x_j^{(i)}|y)}{\sum_{y=1}^{k} q^{t-1}(y) \prod_{j=1}^{d} q_j^{t-1}(x_j^{(i)}|y)}$$

2. Calculate the new parameter values:

$$q^t(y) = \frac{1}{n} \sum_{i=1}^{n} \delta(y|i) \qquad q_j^t(x|y) = \frac{\sum_{i:x_j^{(i)}=x} \delta(y|i)}{\sum_i \delta(y|i)}$$

Output: Parameter values $q^T(y)$ and $q^T(x|y)$.

Figure 1: The EM Algorithm for Naive Bayes Models

For example, in Naive Bayes we have $\mathcal{X} = \{-1, +1\}^d$ for some integer d, and $\mathcal{Y} = \{1 \ldots k\}$ for some integer k. The parameter vector $\underline{\theta}$ contains parameters of the form $q(y)$ and $q_j(x|y)$. The model is

$$p(\underline{x}, y; \underline{\theta}) = q(y) \prod_{j=1}^{d} q_j(x|y)$$

- We use Ω to refer to the set of all valid parameter settings in the model.

  For example, in Naive Bayes Ω contains all parameter vectors such that $q(y) \geq 0$, $\sum_y q(y) = 1$, $q_j(x|y) \geq 0$, and $\sum_x q_j(x|y) = 1$ (i.e., the usual constraints on parameters in a Naive Bayes model).

- We have a training set consisting of examples $x^{(i)}$ for $i = 1 \ldots n$, where each $x^{(i)} \in \mathcal{X}$.

- The log-likelihood function is then

$$L(\underline{\theta}) = \sum_{i=1}^{n} \log p(x^{(i)}; \underline{\theta}) = \sum_{i=1}^{n} \log \sum_{y \in \mathcal{Y}} p(x^{(i)}, y; \underline{\theta})$$

- The maximum likelihood estimates are

$$\underline{\theta}^* = \arg\max_{\underline{\theta} \in \Omega} L(\underline{\theta})$$

In general, finding the maximum-likelihood estimates in this setting is intractable (the function $L(\underline{\theta})$ is a difficult function to optimize, because it contains many local optima).

The EM algorithm is an iterative algorithm that defines parameter settings $\underline{\theta}^0, \underline{\theta}^1, \ldots, \underline{\theta}^T$ (again, see figure 2). The algorithm is driven by the updates

$$\underline{\theta}^t = \arg\max_{\underline{\theta} \in \Omega} Q(\underline{\theta}, \underline{\theta}^{t-1})$$

for $t = 1 \ldots T$. The function $Q(\underline{\theta}, \underline{\theta}^{t-1})$ is defined as

$$Q(\underline{\theta}, \underline{\theta}^{t-1}) = \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} \delta(y|i) \log p(x^{(i)}, y; \underline{\theta}) \tag{11}$$

where

$$\delta(y|i) = p(y|x^{(i)}; \underline{\theta}^{t-1}) = \frac{p(x^{(i)}, y; \underline{\theta}^{t-1})}{\sum_{y \in \mathcal{Y}} p(x^{(i)}, y; \underline{\theta}^{t-1})}$$

Thus as described before in the EM algorithm for Naive Bayes, the basic idea is to fill in the $\delta(y|i)$ values using the conditional distribution under the previous parameter values (i.e., $\delta(y|i) = p(y|x^{(i)}; \underline{\theta}^{t-1})$).

Inputs: Sets $\mathcal{X}$ and $\mathcal{Y}$, where $\mathcal{Y}$ is a finite set (e.g., $\mathcal{Y} = \{1, 2, \ldots k\}$ for some integer k). A model $p(x, y; \underline{\theta})$ that assigns a probability to each (x, y) such that $x \in \mathcal{X}$, $y \in \mathcal{Y}$, under parameters $\underline{\theta}$. A set of Ω of possible parameter values in the model. A training sample $x^{(i)}$ for $i \in \{1 \ldots n\}$, where each $x^{(i)} \in \mathcal{X}$. A parameter T specifying the number of iterations of the algorithm.

Initialization: Set $\underline{\theta}^0$ to some initial value in the set Ω (e.g., a random initial value under the constraint that $\underline{\theta} \in \Omega$).

Algorithm:
For $t = 1 \ldots T$

$$\underline{\theta}^t = \arg\max_{\underline{\theta} \in \Omega} Q(\underline{\theta}, \underline{\theta}^{t-1})$$

where

$$Q(\underline{\theta}, \underline{\theta}^{t-1}) = \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} \delta(y|i) \log p(x^{(i)}, y; \underline{\theta})$$

and

$$\delta(y|i) = p(y|x^{(i)}; \underline{\theta}^{t-1}) = \frac{p(x^{(i)}, y; \underline{\theta}^{t-1})}{\sum_{y \in \mathcal{Y}} p(x^{(i)}, y; \underline{\theta}^{t-1})}$$

Output: Parameters $\underline{\theta}^T$.

Figure 2: The EM Algorithm in General Form

Remark: Relationship to Maximum-Likelihood Estimation for Fully Observed Data. For completeness, figure 3 shows the algorithm for maximum-likelihood estimation in the case of fully observed data: that is, the case where labels $y^{(i)}$ are also present in the training data. In this case we simply set

$$\underline{\theta}^* = \arg\max_{\underline{\theta} \in \Omega} \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} \delta(y|i) \log p(x^{(i)}, y; \underline{\theta}) \tag{12}$$

where $\delta(y|i) = 1$ if $y = y^{(i)}$, 0 otherwise.

Crucially, note the similarity between the optimization problems in Eq. 12 and Eq. 11. In many cases, *if the problem in Eq. 12 is easily solved (e.g., it has a closed-form solution), then the problem in Eq. 11 is also easily solved.*

Inputs: Sets $\mathcal{X}$ and $\mathcal{Y}$, where $\mathcal{Y}$ is a finite set (e.g., $\mathcal{Y} = \{1, 2, \ldots k\}$ for some integer k). A model $p(x, y; \underline{\theta})$ that assigns a probability to each (x, y) such that $x \in \mathcal{X}$, $y \in \mathcal{Y}$, under parameters $\underline{\theta}$. A set of Ω of possible parameter values in the model. A training sample $(x^{(i)}, y^{(i)})$ for $i \in \{1 \ldots n\}$, where each $x^{(i)} \in \mathcal{X}$, $y^{(i)} \in \mathcal{Y}$.

Algorithm: Set $\underline{\theta}^* = \arg\max_{\underline{\theta} \in \Omega} L(\underline{\theta})$ where

$$L(\underline{\theta}) = \sum_{i=1}^{n} \log p(x^{(i)}, y^{(i)}; \underline{\theta}) = \sum_{i=1}^{n} \sum_{y \in \mathcal{Y}} \delta(y|i) \log p(x^{(i)}, y; \underline{\theta})$$

and

$$\delta(y|i) = 1 \text{ if } y = y^{(i)}, 0 \text{ otherwise}$$

Output: Parameters $\underline{\theta}^*$.

Figure 3: Maximum-Likelihood Estimation with Fully Observed Data

6.2 Guarantees for the Algorithm

We now turn to guarantees for the algorithm in figure 2. The first important theorem (which we will prove very shortly) is as follows:

Theorem 4 *For any $\underline{\theta}, \underline{\theta}^{t-1} \in \Omega$, $L(\underline{\theta}) - L(\underline{\theta}^{t-1}) \geq Q(\underline{\theta}, \underline{\theta}^{t-1}) - Q(\underline{\theta}^{t-1}, \underline{\theta}^{t-1})$.*

The quantity $L(\underline{\theta}) - L(\underline{\theta}^{t-1})$ is the amount of progress we make when moving from parameters $\underline{\theta}^{t-1}$ to $\underline{\theta}$. The theorem states that this quantity is lower-bounded by $Q(\underline{\theta}, \underline{\theta}^{t-1}) - Q(\underline{\theta}^{t-1}, \underline{\theta}^{t-1})$.

Theorem 4 leads directly to the following theorem, which states that the likelihood is non-decreasing at each iteration:

Theorem 5 *For $t = 1 \ldots T$, $L(\underline{\theta}^t) \geq L(\underline{\theta}^{t-1})$.*

Proof: By the definitions in the algorithm, we have

$$\underline{\theta}^t = \arg\max_{\underline{\theta} \in \Omega} Q(\underline{\theta}, \underline{\theta}^{t-1})$$

It follows immediately that

$$Q(\underline{\theta}^t, \underline{\theta}^{t-1}) \geq Q(\underline{\theta}^{t-1}, \underline{\theta}^{t-1})$$

(because otherwise $\underline{\theta}^t$ would not be the $\arg\max$), and hence

$$Q(\underline{\theta}^t, \underline{\theta}^{t-1}) - Q(\underline{\theta}^{t-1}, \underline{\theta}^{t-1}) \geq 0$$

But by theorem 4 we have

$$L(\underline{\theta}^t) - L(\underline{\theta}^{t-1}) \geq Q(\underline{\theta}^t, \underline{\theta}^{t-1}) - Q(\underline{\theta}^{t-1}, \underline{\theta}^{t-1})$$

and hence $L(\underline{\theta}^t) - L(\underline{\theta}^{t-1}) \geq 0$. $\square$

Theorem 5 states that the log-likelihood is non-decreasing: but this is a relatively weak guarantee; for example, we would have $L(\underline{\theta}^t) - L(\underline{\theta}^{t-1}) \geq 0$ for $t = 1 \ldots T$ for the trivial definition $\underline{\theta}^t = \underline{\theta}^{t-1}$ for $t = 1 \ldots T$. However, under relatively mild conditions, it can be shown that in the limit as T goes to ∞, the EM algorithm does actually converge to a local optimum of the log-likelihood function $L(\underline{\theta})$. The proof of this is beyond the scope of this note; one reference is Wu, 1983, *On the Convergence Properties of the EM Algorithm*.

To complete this section, we give a proof of theorem 4. The proof depends on a basic property of the $\log$ function, namely that it is concave:

Remark: Concavity of the $\log$ **function.** The $\log$ function is concave. More explicitly, for any values $x_1, x_2, \ldots, x_k$ where each $x_i > 0$, and for any values $\alpha_1, \alpha_2, \ldots, \alpha_k$ where $\alpha_i \geq 0$ and $\sum_i \alpha_i = 1$,

$$\log\left(\sum_i \alpha_i x_i\right) \geq \sum_i \alpha_i \log x_i$$

The proof is then as follows:

$$\begin{aligned}
L(\underline{\theta}) - L(\underline{\theta}^{t-1}) &= \sum_{i=1}^{n} \log \frac{\sum_y p(x^{(i)}, y; \underline{\theta})}{\sum_y p(x^{(i)}, y; \underline{\theta}^{t-1})} \\
&= \sum_{i=1}^{n} \log \sum_y \left(\frac{p(x^{(i)}, y; \underline{\theta})}{p(x^{(i)}; \underline{\theta}^{t-1})} \right) \\
&= \sum_{i=1}^{n} \log \sum_y \left(\frac{p(y|x^{(i)}; \underline{\theta}^{t-1}) \times p(x^{(i)}, y; \underline{\theta})}{p(y|x^{(i)}; \underline{\theta}^{t-1}) \times p(x^{(i)}; \underline{\theta}^{t-1})} \right) \qquad (13) \\
&= \sum_{i=1}^{n} \log \sum_y \left(\frac{p(y|x^{(i)}; \underline{\theta}^{t-1}) \times p(x^{(i)}, y; \underline{\theta})}{p(x^{(i)}, y; \underline{\theta}^{t-1})} \right) \\
&\geq \sum_{i=1}^{n} \sum_y p(y|x^{(i)}; \underline{\theta}^{t-1}) \log \left(\frac{p(x^{(i)}, y; \underline{\theta})}{p(x^{(i)}, y; \underline{\theta}^{t-1})} \right) \qquad (14)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{i=1}^{n} \sum_{y} p(y|x^{(i)}; \underline{\theta}^{t-1}) \log p(x^{(i)}, y; \underline{\theta}) - \sum_{i=1}^{n} \sum_{y} p(y|x^{(i)}; \underline{\theta}^{t-1}) \log p(x^{(i)}, y; \underline{\theta}^{t-1}) \\
&= Q(\underline{\theta}, \underline{\theta}^{t-1}) - Q(\underline{\theta}^{t-1}, \underline{\theta}^{t-1}) \qquad (15)
\end{aligned}
$$

The proof uses some simple algebraic manipulations, together with the property that the $\log$ function is concave. In Eq. 13 we multiply both numerator and denominator by $p(y|x^{(i)}; \underline{\theta}^{t-1})$. To derive Eq. 14 we use the fact that the $\log$ function is concave: this allows us to pull the $p(y|x^{(i)}; \underline{\theta}^{t-1})$ outside the $\log$.

A Proof of Theorem 1

We now prove the result in theorem 1. Our first step is to re-write the log-likelihood function in a way that makes direct use of "counts" taken from the training data:

$$
\begin{aligned}
L(\underline{\theta}) &= \sum_{i=1}^{n} \log q(y_i) + \sum_{i=1}^{n} \sum_{j=1}^{d} \log q_j(x_{i,j}|y_i) \\
&= \sum_{y \in \mathcal{Y}} \text{count}(y) \log q(y) \\
&\quad + \sum_{j=1}^{d} \sum_{y \in \mathcal{Y}} \sum_{x \in \{-1,+1\}} \text{count}_j(x|y) \log q_j(x|y) \qquad (16)
\end{aligned}
$$

where as before

$$\text{count}(y) = \sum_{i=1}^{n} [[y^{(i)} = y]]$$

$$\text{count}_j(x|y) = \sum_{i=1}^{n} [[y_i = y \text{ and } x_j^{(i)} = x]]$$

Eq. 16 follows intuitively because we are simply counting up the number of times each parameter of the form $q(y)$ or $q_j(x|y)$ appears in the sum

$$\sum_{i=1}^{n} \log q(y_i) + \sum_{i=1}^{n} \sum_{j=1}^{d} \log q_j(x_{i,j}|y_i)$$

To be more formal, consider the term

$$\sum_{i=1}^{n} \log q(y^{(i)})$$

We can re-write this as

$$\begin{aligned}
\sum_{i=1}^{n} \log q(y^{(i)}) &= \sum_{i=1}^{n} \sum_{y=1}^{k} [[y^{(i)} = y]] \log q(y) \\
&= \sum_{y=1}^{k} \sum_{i=1}^{n} [[y^{(i)} = y]] \log q(y) \\
&= \sum_{y=1}^{k} \log q(y) \sum_{i=1}^{n} [[y^{(i)} = y]] \\
&= \sum_{y=1}^{k} (\log q(y)) \times \text{count}(y)
\end{aligned}$$

The identity

$$\sum_{i=1}^{n} \sum_{j=1}^{d} \log q_j(x_{i,j}|y_i) = \sum_{j=1}^{d} \sum_{y \in \mathcal{Y}} \sum_{x \in \{-1,+1\}} \text{count}_j(x|y) \log q_j(x|y)$$

can be shown in a similar way.

Now consider again the expression in Eq. 16:

$$\sum_{y \in \mathcal{Y}} \text{count}(y) \log q(y) + \sum_{j=1}^{d} \sum_{y \in \mathcal{Y}} \sum_{x \in \{-1,+1\}} \text{count}_j(x|y) \log q_j(x|y)$$

Consider first maximization of this function with respect to the $q(y)$ parameters. It is easy to see that the term

$$\sum_{j=1}^{d} \sum_{y \in \mathcal{Y}} \sum_{x \in \{-1,+1\}} \text{count}_j(x|y) \log q_j(x|y)$$

does not depend on the $q(y)$ parameters at all. Hence to pick the optimal $q(y)$ parameters, we need to simply maximize

$$\sum_{y \in \mathcal{Y}} \text{count}(y) \log q(y)$$

subject to the constraints $q(y) \geq 0$ and $\sum_{y=1}^{k} q(y) = 1$. But by theorem 2, the values for $q(y)$ which maximize this expression under these constraints is simply

$$q(y) = \frac{\text{count}(y)}{\sum_{y=1}^{k} \text{count}(y)} = \frac{\text{count}(y)}{n}$$

By a similar argument, we can maximize each term of the form

$$\sum_{x \in \{-1,+1\}} \text{count}_j(x|y) \log q_j(x|y)$$

for a given $j \in \{1 \ldots k\}$, $y \in \{1 \ldots k\}$ separately. Applying theorem 2 gives

$$q_j(x|y) = \frac{\text{count}_j(x|y)}{\sum_{x \in \{-1,+1\}} \text{count}_j(x|y)}$$