A Technical Introduction to Statistical Natural Language Processing

Jacob Eisenstein

January 8, 2017

Contents

[1]This section follows closely from J&M 2009

Preface

This text is built from the notes that I use for teaching Georgia Tech's undergraduate and graduate courses on natural language processing, CS 4650 and 7650. There are several other good resources (e.g., Manning and Schütze, 1999; Jurafsky and Martin, 2009; Smith, 2011; Figueiredo et al., 2013; Collins, 2013), but for various reasons I wanted to create something of my own.

The text assumes familiarity with basic linear algebra, and with calculus through Lagrange multipliers. It includes a refresher on probability, but some previous exposure would be helpful. An introductory course on the analysis of algorithms is also assumed; in particular, the reader should be familiar with asymptotic analysis of the time and memory costs of algorithms, and should have seen dynamic programming. No prior background in machine learning or linguistics is assumed, and even students with background in machine learning should be sure to read the introductory chapters, since the notation used in natural language processing is different from typical machine learning presentations, due to the emphasis on structure prediction in applications of machine learning to language. Throughout the book, advanced material is marked with an asterisk, and can be safely skipped.

The notes focus on what I view as a core subset of the field of natural language processing, unified by the concepts of linear models and structure prediction. A remarkable thing about the field of natural language processing is that so many problems in language technology can be solved by a small number of methods. These notes focus on the following methods:

Search algorithms shortest path, Viterbi, CKY, minimum spanning tree, shift-reduce, integer linear programming, dual decomposition (maybe), beam search.

Learning algorithms Naïve Bayes, logistic regression, perceptron, expectation-maximization, matrix factorization, backpropagation.

The goal of this text is to teach how these methods work, and how they can be applied to problems that arise in the computer processing of natural language: document classification, word sense disambiguation, sequence labeling (part-of-speech tagging and named entity recognition), parsing, coreference resolution, relation extraction, discourse analysis,

and, to a limited degree, language modeling and machine translation. Because proper application of these techniques requires understanding the underlying linguistic phenomena, the notes also include chapters on the foundations of morphology, syntactic parts of speech, context-free grammar, semantics, and discourse; however, for a detailed understanding of these topics, a full-fledged linguistics textbook should be consulted (e.g., Akmajian et al., 2010; Fromkin et al., 2013).

-Jacob Eisenstein, January 8, 2017

Notation

w_n	word token at position n
$\boldsymbol{x}^{(i)}$	a (column) vector of feature counts for instance i, often word counts
$\boldsymbol{x}_{i:j}$	elements i through j (inclusive) of a vector $\boldsymbol{x}$
$y^{(i)}$	the label for instance i
$\hat{y}$	a predicted label
$\boldsymbol{y}$	a vector of labels across all instances
$\mathcal{Y}$	the set of all possible labels
K	number of possible labels $K = \#\lvert\mathcal{Y}\rvert$
$\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$	feature vector for instance i with label $y^{(i)}$
$\boldsymbol{\theta}$	a (column) vector of weights
$\boldsymbol{\theta}^\top$	the transpose of the vector $\boldsymbol{\theta}$
$\Pr(A)$	probability of event A
$\mathrm{p}_B(b)$	the marginal probability of random variable B taking value b
$\mathcal{T}(\boldsymbol{w})$	the set of possible tag sequences for the word sequence $\boldsymbol{w}$
$\lozenge$	the start tag
$\blacklozenge$	the stop tag
$\square$	the start token
$\blacksquare$	the stop token
λ	the amount of regularization

Part I

Words, bags of words, and features

Chapter 1

Linear classification and features

Suppose you want to build a spam detector, in which each document is classified as "spam" or "ham." How would you do it, using only the text in the email?

One solution is to represent document i as a column vector of word counts: $\boldsymbol{x}^{(i)} = [0\ 1\ 1\ 0\ 0\ 2\ 0\ 1\ 13\ 0 \ldots]^\top$, where $x_{i,j}$ is the count of word j in document i. Suppose the size of the vocabulary is V, so that the length of $\boldsymbol{x}^{(i)}$ is also V. The object $\boldsymbol{x}^{(i)}$ is a vector, but colloquially we call it a **bag of words**, because it includes only information about the count of each word, and not the order in which they appear.

We've thrown out grammar, sentence boundaries, paragraphs — everything but the words! But this could still work. If you see the word *free*, is it spam or ham? How about *Bayesian*? One approach would be to define a "spamminess" score for every word in the dictionary, and then just add them up. These scores are called **weights**, written $\boldsymbol{\theta}$, and we'll spend a lot of time talking about where they come from.

But for now, let's generalize: suppose we want to build a multi-way classifier to distinguish stories about sports, celebrities, music, and business. Each label $y^{(i)}$ is a member of a set of K possible labels $\mathcal{Y}$. Our goal is to predict a label $\hat{y}^{(i)}$, given the bag of words $\boldsymbol{x}^{(i)}$, using the weights $\boldsymbol{\theta}$. We'll do this using a vector inner product between the weights $\boldsymbol{\theta}$ and a **feature vector** $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$. As the notation suggests, the feature vector is constructed by combining $\boldsymbol{x}^{(i)}$ and $y^{(i)}$. For example, feature j might be,

$$f_j(\boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 1, & \text{if}(\textit{freeee} \in \boldsymbol{x}^{(i)}) \wedge (y^{(i)} = \textsc{Spam}) \\ 0, & \text{otherwise} \end{cases} \tag{1.1}$$

For any pair $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$, we then define $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$ as,

$$\boldsymbol{f}(\boldsymbol{x}, Y = 0) = [\boldsymbol{x}^\top, \underbrace{0, 0, \ldots, 0}_{V \times (K-1)}]^\top \tag{1.2}$$

$$\boldsymbol{f}(\boldsymbol{x}, Y = 1) = [\underbrace{0, 0, \ldots, 0}_{V}, \boldsymbol{x}^\top, \underbrace{0, 0, \ldots, 0}_{V \times (K-2)}]^\top \tag{1.3}$$

$$\boldsymbol{f}(\boldsymbol{x}, Y = 2) = [\underbrace{0, 0, \ldots, 0}_{2 \times V}, \boldsymbol{x}^\top, \underbrace{0, 0, \ldots, 0}_{V \times (K-3)}]^\top \tag{1.4}$$

$$\boldsymbol{f}(\boldsymbol{x}, Y = K) = [\underbrace{0, 0, \ldots, 0}_{V \times (K-1)}, \boldsymbol{x}^\top]^\top, \tag{1.5}$$

where $\underbrace{0, 0, \ldots, 0}_{V \times (K-1)}$ is a column vector of $V \times (K-1)$ zeros. This arrangement is shown in Figure 1.1. This notation may seem like a strange choice, but in fact it helps to keep things simple. Given a vector of weights, $\boldsymbol{\theta} \in \mathbb{R}^{V \times K}$, we can now compute the inner product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$. This inner product gives a scalar measure of the score for label y, given observations $\boldsymbol{x}$. For any document $\boldsymbol{x}^{(i)}$, we predict the label $\hat{y}$ as

$$\hat{y} = \underset{y}{\text{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{1.6}$$

This inner product is the fundamental equation for linear classification, and it is the reason we prefer the feature function notation $\boldsymbol{f}(\boldsymbol{x}, y)$. The notation gives a clean separation between the **data** ($\boldsymbol{x}$ and y) and the **parameters**, which are expressed by the single vector of weights, $\boldsymbol{\theta}$. As we will see in later chapters, this notation also generalizes nicely to **structured output spaces**, in which the space of labels $\mathcal{Y}$ is very large, and we want to model shared substructure between labels.

Often we'll add an **offset** feature at the end of $\boldsymbol{x}$, which is always 1; we then have to also add an extra zero to each of the zero vectors. This gives the entire feature vector $\boldsymbol{f}(\boldsymbol{x}, y)$ a length of $(V + 1) \times K$. The weight associated with this offset feature can be thought of as a "bias" for each label. For example, if we expect most documents to be spam, then the weight for the offset feature for $Y = \text{spam}$ should be larger than the weight for the offset feature for $Y = \text{ham}$.

Returning to the weights $\boldsymbol{\theta}$ — where do they come from? As already suggested, we could set the weights by hand. If we wanted to distinguish, say, English from Spanish, we could use English and Spanish dictionaries, and set the weight to one for each word that

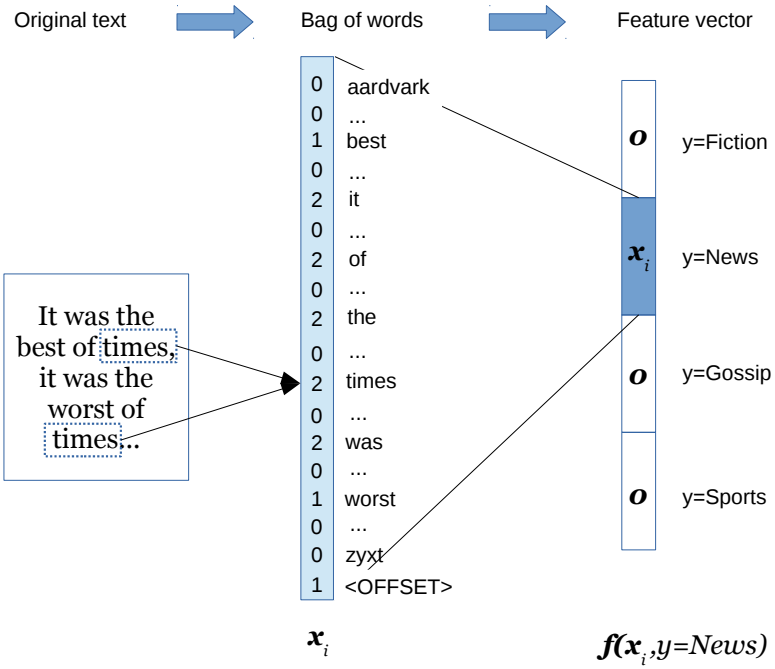

Figure 1.1: The bag-of-words and feature vector representations, for a hypothetical text classification task.

appears in the associated dictionary. For example,[1]

$$\theta_{(E,bicycle)} = 1 \qquad \theta_{(S,bicycle)} = 0$$
$$\theta_{(E,bicicleta)} = 0 \qquad \theta_{(S,bicicleta)} = 1$$
$$\theta_{(E,con)} = 1 \qquad \theta_{(S,con)} = 1$$
$$\theta_{(E,ordinateur)} = 0 \qquad \theta_{(S,ordinateur)} = 0.$$

Similarly, if we want to distinguish positive and negative sentiment, we could use positive and negative *sentiment lexicons*, which are defined by expert psychologists (Tausczik and Pennebaker, 2010). You'll try this in Problem Set 1.

But it is usually not easy to set classification weights by hand. Instead, we will learn them from data. For example, email users manually label thousands of messages as "spam" or "not spam"; newspapers label their own articles as "business" or "fashion." Such **instance labels** are a typical form of labeled data that we will encounter in NLP. In **supervised machine learning** , we use instance labels to automatically set the weights for a classifier. An important tool for this is probability.

[1] In this notation, each tuple (language, word) indexes an element in $\boldsymbol{\theta}$, which remains a vector.

1.1 Review of basic probability

Probability theory provides a way to reason about random events. The sorts of random events that are typically used to explain probability theory include coin flips, card draws, and the weather. It may seem odd to think about the choice of a word as akin to the flip of a coin, particularly if you are the type of person to choose words carefully. But random or not, language has proven to be extremely difficult to model deterministically. Probability offers a powerful tool for modeling and manipulating linguistic data, which we will use repeatedly throughout this course.[2]

Probability can be thought of in terms of **random outcomes** : for example, a single coin flip has two possible outcomes, heads or tails. The set of possible outcomes is the **sample space** , and a subset of the **sample space** is an **event** . For a sequence of two coin flips, there are four possible outcomes, $\{HH, HT, TH, TT\}$, representing the ordered sequences heads-head, heads-tails, tails-heads, and tails-tails. The event of getting exactly one head includes two outcomes: $\{HT, TH\}$.

Formally, a probability is a function from events to the interval between zero and one: $\Pr : \mathcal{F} \rightarrow [0, 1]$, where $\mathcal{F}$ is the set of possible events. An event that is certain has probability one; an event that is impossible has probability zero. For example, the probability of getting less than three heads on two coin flips is one. Each outcome is also an event (a set with exactly one element), and for two flips of a fair coin, the probability of each outcome is,

$$\Pr(\{HH\}) = \Pr(\{HT\}) = \Pr(\{TH\}) = \Pr(\{TT\}) = \frac{1}{4}. \tag{1.7}$$

Probabilities of event combinations

Because events are **sets** of outcomes, we can use set theoretic operations such complement, intersection, and unions to reason about the probabilities of various event combinations.

For any event A, there is a **complement** $\neg A$, such that:

- The union $A \cup \neg A$ covers the entire sample space, and $\Pr(A \cup \neg A) = 1$;
- The intersection $A \cap \neg A = \varnothing$ is the empty set, and $\Pr(A \cap \neg A) = 0$.

In the coin flip example, the event of obtaining a single head on two flips corresponds to the set of outcomes $\{HT, TH\}$; the complement event includes the other two outcomes, $\{TT, HH\}$.

[2] A good introduction to probability theory is offered by Manning and Schütze (1999), which helped to motivate this section. For more detail, Sharon Goldwater provides another useful reference, `http://homepages.inf.ed.ac.uk/sgwater/teaching/general/probability.pdf`.

Probabilities of disjoint events

In general, when two events have an empty intersection, $A \cap B = \varnothing$, they are said to be **disjoint** . The probability of the union of two disjoint events is equal to the sum of their probabilities,

$$A \cap B = \varnothing \quad \Rightarrow \quad \Pr(A \cup B) = \Pr(A) + \Pr(B). \tag{1.8}$$

This is the **third axiom of probability** , and can be generalized to any countable sequence of disjoint events.

In the coin flip example, we can use this axiom to derive the probability of the event of getting a single head on two flips. This event is the set of outcomes $\{HT, TH\}$, which is the union of two simpler events, $\{HT, TH\} = \{HT\} \cup \{TH\}$. The events $\{HT\}$ and $\{TH\}$ are disjoint. Therefore,

$$\Pr(\{HT, TH\}) = \Pr(\{HT\} \cup \{TH\}) = \Pr(\{HT\}) + \Pr(\{TH\}) \tag{1.9}$$

$$= \frac{1}{4} + \frac{1}{4} = \frac{1}{2}. \tag{1.10}$$

For events that are not disjoint, it is still possible to compute the probability of their union:

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B). \tag{1.11}$$

This can be derived from the third axiom of probability. First, consider an event that includes all outcomes in B that are not in A, which we can write as $B - (A \cap B)$. By construction, this event is disjoint from A.[3] We can therefore apply the additive rule,

$$\Pr(A \cup B) = \Pr(A) + \Pr(B - (A \cap B)) \tag{1.12}$$

$$\Pr(B) = \Pr(B - (A \cap B)) + \Pr(A \cap B) \tag{1.13}$$

$$\Pr(B - (A \cap B)) = \Pr(B) - \Pr(A \cap B) \tag{1.14}$$

$$\Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B). \tag{1.15}$$

Law of total probability

A set of events $\mathcal{B} = \{B_1, B_2, \ldots, B_N\}$ is a **partition** of the sample space iff each pair of events is disjoint ($B_i \cap B_j = \varnothing$), and the union of the events is the entire sample space. The law of total probability states that we can **marginalize** over these events as follows,

$$\Pr(A) = \sum_{B_n \in \mathcal{B}} \Pr(A \cap B_n). \tag{1.16}$$

Note for any event B, the union $B \cup \neg B$ forms a partition of the sample space. Therefore, an important special case of the law of total probability is,

$$\Pr(A) = \Pr(A \cap B) + \Pr(A \cap \neg B). \tag{1.17}$$

[3] [todo: add figure]

Conditional probability and Bayes' rule

A **conditional probability** is an expression like $\Pr(A \mid B)$, which is the probability of the event A, assuming that event B happens too. For example, we may be interested in the probability of a randomly selected person answering the phone by saying *hello*, conditioned on that person being a speaker of English. We define conditional probability as the ratio,

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)} \tag{1.18}$$

The **chain rule** states that $\Pr(A \cap B) = \Pr(A \mid B) \times \Pr(B)$, which is just a simple rearrangement of terms from Equation 1.18. We can apply the chain rule repeatedly:

$$\begin{aligned}\Pr(A \cap B \cap C) &= \Pr(A \mid B \cap C) \times \Pr(B \cap C) \\ &= \Pr(A \mid B \cap C) \times \Pr(B \mid C) \times \Pr(C)\end{aligned}$$

Bayes' rule (sometimes called Bayes' law or Bayes' theorem) gives us a way to convert between $\Pr(A \mid B)$ and $\Pr(B \mid A)$. It follows from the chain rule:

$$\Pr(A \mid B) = \frac{\Pr(A \cap B)}{\Pr(B)} = \frac{\Pr(B \mid A) \times \Pr(A)}{\Pr(B)} \tag{1.19}$$

The terms in Bayes rule have specialized names, which we will occasionally use:

- $\Pr(A)$ is the **prior** , since it is the probability of event A without knowledge about whether B happens or not.
- $\Pr(B \mid A)$ is the **likelihood** , the probability of event B given that event A has occurred.
- $\Pr(A \mid B)$ is the **posterior** , since it is the probability of event A with knowledge that B has occurred.

Example Manning and Schütze (1999) have a nice example of Bayes' rule (sometimes called Bayes Law) in a linguistic setting. (This same example is usually framed in terms of tests for rare diseases.) Suppose one is interested in a rare syntactic construction, such as **parasitic gaps**, which occur on average once in 100,000 sentences. Here is an example:

(1.1) *Which class did you attend __ without registering for __?*

Lana Linguist has developed a complicated pattern matcher that attempts to identify sentences with parasitic gaps. It's pretty good, but it's not perfect:

- If a sentence has a parasitic gap, the pattern matcher will find it with probability 0.95. (Skipping ahead, this is the **recall**; the **false negative rate** is defined as one minus the recall.)
- If the sentence doesn't have a parasitic gap, the pattern matcher will wrongly say it does with probability 0.005. (This is the **false positive rate**. The **precision** is defined as one minus the false positive rate.)

Suppose that Lana's pattern matcher says that a sentence contains a parasitic gap. What is the probability that this is true?

Let G be the event of a sentence having a parasitic gap, and T be the event of the test being positive. We are interested in the probability of a sentence having a parasitic gap given that the test is positive. This is the conditional probability $\Pr(G \mid T)$, and we can compute it from Bayes' rule:

$$\Pr(G \mid T) = \frac{\Pr(T \mid G) \times \Pr(G)}{\Pr(T)}. \tag{1.20}$$

We already know both terms in the numerator: $\Pr(T \mid G)$ is the recall, which is 0.95; $\Pr(G)$ is the prior, which is 10^{-5}.

We are not given the denominator, but we can compute it by using some of the tools that we have developed in this section. We first apply the law of total probability, using the partition $\{G, \neg G\}$:

$$\Pr(T) = \Pr(T \cap G) + \Pr(T \cap \neg G). \tag{1.21}$$

This says that the probability of the test being positive is the sum of the probability of a **true positive** ($T \cap G$) and the probability of a **false positive** ($T \cap \neg G$). Next, we can compute the probability of each of these events using the chain rule:

$$\Pr(T \cap G) = \Pr(T \mid G) \times \Pr(G) = 0.95 \times 10^{-5} \tag{1.22}$$

$$\Pr(T \cap \neg G) = \Pr(T \mid G) \times \Pr(\neg G) = 0.005 \times (1 - 10^{-5}) \approx 0.005 \tag{1.23}$$

$$\Pr(T) = \Pr(T \cap G) + \Pr(T \cap \neg G) \tag{1.24}$$

$$= 0.95 \times 10^{-5} + 0.005 \approx 0.005. \tag{1.25}$$

We now return to Bayes' rule to compute the desired posterior probability,

$$\Pr(G \mid T) = \frac{\Pr(T \mid G) \Pr(G)}{\Pr(T)} \tag{1.26}$$

$$= \frac{0.95 \times 10^{-5}}{0.95 \times 10^{-5} + 0.005 \times (1 - 10^{-5})} \tag{1.27}$$

$$\approx 0.002. \tag{1.28}$$

Lana's pattern matcher is very accurate, with false positive and false negative rates below 5%. Yet the extreme rarity of this phenomenon means that a positive result from the detector is most likely to be wrong.

Independence

Two events are independent if the probability of their intersection is equal to the product of their probabilities: $\Pr(A \cap B) = \Pr(A) \times \Pr(B)$. For example, for two flips of a fair coin, the probability of getting heads on the first flip is independent of the probability of getting heads on the second flip. We can prove this by using the additive axiom defined above:

$$\Pr(\{HT, HH\}) = \Pr(HT) + \Pr(HH) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2} \tag{1.29}$$

$$\Pr(\{HH, TH\}) = \Pr(HH) + \Pr(TH) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2} \tag{1.30}$$

$$\Pr(\{HT, HH\}) \times \Pr(\{HH, TH\}) = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} \tag{1.31}$$

$$\Pr(\{HT, HH\} \cap \{HH, TH\}) = \Pr(HH) = \frac{1}{4} \tag{1.32}$$

$$= \Pr(\{HT, HH\}) \times \Pr(\{HH, TH\}). \tag{1.33}$$

Independence will play a key role in the discussion of probabilistic classification later in this chapter.

If $\Pr(A \cap B \mid C) = \Pr(A \mid C) \times \Pr(B \mid C)$, then the events A and B are **conditionally independent** , written $A \perp B \mid C$.

Random variables

Random variables are functions of events. Formally, we will treat random variables as functions from events to the space $\mathbb{R}^n$, where $\mathbb{R}$ is the set of real numbers. This general notion subsumes a number of different types of random variables:

- **Indicator random variables** are functions from events to the set $\{0, 1\}$. In the coin flip example, we can define X as an indicator random variable, for whether the coin has come up heads on at least one flip. This would include the outcomes $\{HH, HT, TH\}$. The event probability $\Pr(X = 1)$ is the sum of the probabilities of these outcomes, $\Pr(X = 1) = \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4}$.
- A **discrete random variable** is a function from events to a countable subset of $\mathbb{R}$. Consider the coin flip example: the number of heads, X, can be viewed as a discrete random variable, $X \in 0, 1, 2$. The event probability $\Pr(X = 1)$ can again be computed as the sum of the probabilities of the events in which there is one head, $\{HT, TH\}$, giving $\Pr(X = 1) = \frac{1}{2}$.

Each possible value of a random variable is associated with a subset of the sample space. In the coin flip example, $X = 0$ is associated with the event $\{TT\}$, $X = 1$ is associated with the event $\{HT, TH\}$, and $X = 2$ is associated with the event $\{HH\}$.

Assuming a fair coin, the probabilities of these events are, respectively, $1/4$, $1/2$, and $1/4$. This list of numbers represents the **probability distribution** over X, written p_X, which maps from the possible values of X to the non-negative reals. For a specific value x, we write $\mathrm{p}_X(x)$, which is equal to the event probability $\Pr(X = x)$.[4] The function p_X is called a probability **mass** function (pmf) if X is discrete; it is called a probability **density** function (pdf) if X is continuous. In either case, we have $\int_x \mathrm{p}_X(x)dx = 1$ and $\forall x, \mathrm{p}_X(x) \geq 0$.

Random variables can be combined into **joint probabilities**, e.g., $\mathrm{p}_{A,B}(a, b) = \Pr(A = a \cap B = b)$. Several ideas from event probabilities carry over to probability distributions over random variables:

- We can write a **marginal probability distribution** $\mathrm{p}_A(a) = \sum_b \mathrm{p}_{A,B}(a, b)$.
- We can write a **conditional probability distribution** as $\mathrm{p}_{A|B}(a \mid b) = \frac{\mathrm{p}_{A,B}(a,b)}{\mathrm{p}_B(b)}$.
- Random variables A and B are independent iff $\mathrm{p}_{A,B}(a, b) = \mathrm{p}_A(a) \times \mathrm{p}_B(b)$.

Expectations

Sometimes we want the **expectation** of a function, such as $E[g(x)] = \sum_{x \in \mathcal{X}} g(x)\mathrm{p}(x)$. Expectations are easiest to think about in terms of probability distributions over discrete events:

- If it is sunny, Marcia will eat three ice creams.
- If it is rainy, she will eat only one ice cream.
- There's a 80% chance it will be sunny.
- The expected number of ice creams she will eat is $0.8 \times 3 + 0.2 \times 1 = 2.6$.

If the random variable X is continuous, the sum becomes an integral:

$$E[g(x)] = \int_{\mathcal{X}} g(x)\mathrm{p}(x)dx \tag{1.34}$$

For example, a fast food restaurant in Quebec has a special offer for cold days: they give a 1% discount on poutine for every degree below zero. Assuming they use a thermometer with infinite precision, the expected price would be an integral over all possible temperatures,

$$E[\text{price}(x)] = \int_{\mathcal{X}} \min(1, 1 + x) \times \text{original-price} \times \mathrm{p}(x)dx. \tag{1.35}$$

(Careful readers will note that the restaurant will apparently pay you for taking poutine, if the temperature falls below -100 degrees celsius.)

[4] In general, capital letters (e.g., X) refer to random variables, and lower-case letters (e.g., x) refer to specific values. I will often just write $\mathrm{p}(x)$, when the subscript is clear from context.

1.2 Naïve Bayes

Back to text classification, where we were left wondering how to set the weights $\boldsymbol{\theta}$. Having just reviewed basic probability, we can now take a probabilistic approach to this problem. A Naïve Bayes classifier chooses the weights $\boldsymbol{\theta}$ to maximize the *joint* probability of a labeled dataset, $\text{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N})$, where each tuple $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$ is a labeled instance.

We first need to define the probability $\text{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N})$. We'll do that through a "generative model," which describes a hypothesized stochastic process that has generated the observed data.[5]

- For each document i,
  - draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$
  - draw the vector of counts $\boldsymbol{x}^{(i)} \mid y^{(i)} \sim \text{Multinomial}(\boldsymbol{\phi}_{y^{(i)}})$,

The first line of this generative model is "for each document i", which tells us to treat each document independently: the probability of the whole dataset is equal to the product of the probabilities of each individual document. The observed word counts and document labels are **independent and identically distributed (IID)**.

$$\text{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N}; \boldsymbol{\mu}, \boldsymbol{\phi}) = \prod_{i=1}^{N} \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\mu}, \boldsymbol{\phi}) \tag{1.36}$$

This means that the words in each document are **conditionally independent** given the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$.

The second line indicates $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$, which means that the random variable $y^{(i)}$ is a stochastic draw from a categorical distribution with **parameter** $\boldsymbol{\mu}$. A categorical distribution is just like a weighted die: $\text{p}_{\text{cat}}(y; \boldsymbol{\mu}) = \mu_y$, where μ_y is the probability of the outcome $Y = y$. For example, if $\mathcal{Y} = \{\text{positive}, \text{negative}, \text{neutral}\}$, we might have $\boldsymbol{\mu} = [0.1, 0.7, 0.2]$. We require $\sum_y \mu_y = 1$ and $\forall_y, \mu_y \geq 0$.

The third and final line invokes the **multinomial distribution**, which is only slightly more complex:

$$\text{p}_{\text{mult}}(\boldsymbol{x}; \boldsymbol{\phi}) = \frac{\left(\sum_j^V x_j\right)!}{\prod_j^V x_j!} \prod_j^V \phi_j^{x_j} \tag{1.37}$$

We again require that $\sum_j^V \phi_j = 1$ and $\forall j, \phi_j \geq 0$. The second part of the equation is a product over words, with an exponent for each word; recall that $\phi_j^0 = 1$ for all ϕ_j; this means that the words that have zero count play no role in the overall probability.

[5] We'll see a lot of different generative models in this course. They are a helpful tool because they clearly and explicitly define the assumptions that underly the form of the probability distribution. For a very readable introduction to generative models in statistics, see Blei (2014).

The first part of Equation 1.37 doesn't depend on $\boldsymbol{\phi}$, and can usually be ignored. Can you see why we need the first part at all?[6] We will return to this issue shortly.

We can write $\text{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}; \boldsymbol{\phi})$ to indicate the conditional probability of word counts $\boldsymbol{x}^{(i)}$ given label $y^{(i)}$, with parameter $\boldsymbol{\phi}$, which is equal to $\text{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}})$. By specifying the multinomial distribution, we are working with *multinomial naïve Bayes* (MNB). Why "naïve"? Because the multinomial distribution treats each word token independently: the probability mass function factorizes across the counts.[7] We'll see this more clearly later, when we show how MNB is an example of linear classification.

Another version of Naïve Bayes

Consider a slight modification to the generative story of NB:

- For each document i
  - Draw the label $y^{(i)} \sim \text{Categorical}(\boldsymbol{\mu})$
  - For each word $n \leq D_i$
    - Draw the word $w_{i,n} \sim \text{Categorical}(\boldsymbol{\phi}_{y^{(i)}})$

This is not quite the same model as multinomial Naive Bayes (MNB): it's a product of categorical distributions over words, instead of a multinomial distribution over word counts. This means we would generate the words in order, like $\text{p}_W(\textit{multinomial})\text{p}_W(\textit{Naive})\text{p}_W(\textit{Bayes})$. Formally, this is a model for the joint probability $\text{p}(\boldsymbol{w}, y)$, not $\text{p}(\boldsymbol{x}, y)$.

However, as a classifier, it is identical to MNB. The final probabilities are reduced by a factor corresponding to the normalization term in the multinomial, $\frac{(\sum_j x_j)!}{\prod_j x_j!}$. This means that the the probability for a vector of counts $\boldsymbol{x}$ is larger than the probability for a list of words $\boldsymbol{w}$ that induces the same counts. But this makes sense: there can be many word sequences that correspond to a single vector counts. For example, *man bites dog* and *dog bites man* correspond to an identical count vector, $\{\textit{bites} : 1, \textit{dog} : 1, \textit{man} : 1\}$, and the total number of word orderings for a given count vector $\boldsymbol{x}$ is exactly the ratio $\frac{(\sum_j x_j)!}{\prod_j x_j!}$.

From the perspective of classification, none of this matters, because it has nothing to do with the label y or the parameters $\boldsymbol{\phi}$. The ratio of probabilities between any two labels y_1 and y_2 will be identical in the two models, as will the maximum likelihood estimates for the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$ (defined later).

[6] Technically, a multinomial distribution requires a second parameter, the total number of counts, which in the bag-of-words representation is equal to the number of words in the document.

[7] You can plug in any probability distribution to the generative story and it will still be naïve Bayes, as long as you are making the "naïve" assumption that your features are conditionally independent, given the label. For example, a multivariate Gaussian with diagonal covariance would be naïve in exactly the same sense.

Prediction

The Naive Bayes prediction rule is to choose the label y which maximizes $\text{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi})$:

$$\hat{y} = \underset{y}{\text{argmax}}\, \text{p}(\boldsymbol{x}, y; \boldsymbol{\mu}, \boldsymbol{\phi}) \tag{1.38}$$

$$= \underset{y}{\text{argmax}}\, \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi})\text{p}(y; \boldsymbol{\mu}) \tag{1.39}$$

$$= \underset{y}{\text{argmax}}\, \log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) \tag{1.40}$$

Converting to logarithms makes the notation easier. It doesn't change the prediction rule because the log function is monotonically increasing.

Now we can plug in the probability distributions from the generative story.

$$\log \text{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi}) + \log \text{p}(y; \boldsymbol{\mu}) = \log \left[\frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} \prod_j \phi_{y,j}^{x_j} \right] + \log \mu_y \tag{1.41}$$

$$= \log \frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} + \sum_j x_j \log \phi_{y,j} + \log \mu_y \tag{1.42}$$

$$= k + \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y), \tag{1.43}$$

where

$$\boldsymbol{\theta} = [{\boldsymbol{\theta}^{(1)}}^\top, {\boldsymbol{\theta}^{(2)}}^\top, \ldots, {\boldsymbol{\theta}^{(K)}}^\top]^\top \tag{1.44}$$

$$\boldsymbol{\theta}^{(y)} = [\log \phi_{y,1},\ \log \phi_{y,2},\ \ldots,\ \log \phi_{y,V},\ \log \mu_y]^\top \tag{1.45}$$

$$k = \log \frac{\left(\sum_j x_j\right)!}{\prod_j x_j!} \qquad \text{(This is a constant that we can ignore.)} \tag{1.46}$$

The feature function $\boldsymbol{f}(\boldsymbol{x}, y)$ is a vector of V word counts and an offset, padded by zeros for the labels not equal to y (see equations 1.2-1.5, and Figure 1.1). This construction ensures that the inner product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$ only activates the features whose weights are in $\boldsymbol{\theta}^{(y)}$. These features and weights are all we need to compute the joint log-probability $\log \text{p}(\boldsymbol{x}, y)$ for each y. This is a key point: through this notation, we have converted the problem of computing the log-likelihood for a document-label pair $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$ into the computation of a vector inner product.

Estimation

The parameters of a multinomial distribution have a simple interpretation: they are the expected frequency for each word. Based on this interpretation, it is tempting to set the

parameters empirically, as

$$\phi_{y,j} = \frac{\sum_{i:y^{(i)}=y} x_{i,j}}{\sum_{j'} \sum_{i:y^{(i)}=y} x_{i,j'}} = \frac{\text{count}(y,j)}{\sum_{j'} \text{count}(y,j')} \tag{1.47}$$

This is called a *relative frequency estimator*. It can be justified more rigorously as a *maximum likelihood estimate*.

Our prediction rule in Equation 1.38 is to choose $\hat{y}$ so as to maximize the joint probability $\text{p}(\boldsymbol{x}, y)$. Maximum likelihood estimation proposes to choose the parameters $\boldsymbol{\phi}$ and $\boldsymbol{\mu}$ in much the same way. Specifically, we want to maximize the joint log-likelihood of some **training data**, which consists of a set of annotated examples where we observe both the text and the true label, $\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i \in 1 \ldots N}$. Based on the generative model that we have defined, the log-likelihood is:

$$L = \sum_i \log \text{p}_{\text{mult}}(\boldsymbol{x}_i; \boldsymbol{\phi}_{y^{(i)}}) + \log \text{p}_{\text{cat}}(y^{(i)}; \boldsymbol{\mu}). \tag{1.48}$$

Let's continue to focus on the parameters $\boldsymbol{\phi}$. Since $\text{p}(y)$ is constant in L with respect to these parameters, we can forget it for now,

$$L(\boldsymbol{\phi}) = \sum_i \log \text{p}_{\text{mult}}(\boldsymbol{x}^{(i)}; \boldsymbol{\phi}_{y^{(i)}}) \tag{1.49}$$

$$= \sum_i \log \frac{(\sum_j x_{i,j})!}{\prod_j x_{i,j}!} \prod_j \boldsymbol{\phi}_{y^{(i)},j}^{x_{i,j}} \tag{1.50}$$

$$= \sum_i \log \left[(\sum_j x_{i,j})! \right] - \sum_j \log (x_{i,j}!) + \sum_j x_{i,j} \log \boldsymbol{\phi}_{y^{(i)},j} \tag{1.51}$$

$$\propto \sum_j x_{i,j} \log \boldsymbol{\phi}_{y^{(i)},j}, \tag{1.52}$$

where I have abused notation by writing $\propto$ to indicate that the left side of Equation 1.52 is equal to the right side plus terms that are constant with respect to $\boldsymbol{\phi}$.

We would now like to optimize L, by taking derivatives with respect to $\boldsymbol{\phi}$. But before we can do that, we have to deal with a set of constraints:

$$\forall y, \sum_{j=1}^{V} \phi_{y,j} = 1 \tag{1.53}$$

We'll do this by adding a Lagrange multiplier. Solving separately for each label y, we obtain the resulting Lagrangian,

$$\ell[\boldsymbol{\phi}_y] = \sum_{i:Y^{(i)}=y} \sum_j x_{ij} \log \phi_{y,j} - \lambda(\sum_j \phi_{y,j} - 1) \tag{1.54}$$

We can now differentiate the Lagrangian with respect to the parameter of interest, setting $\frac{\partial \ell}{\partial \boldsymbol{\phi}_{y,j}} = 0$,

$$0 = \sum_{i:Y^{(i)}=y} x_{i,j}/\phi_{y,j} - \lambda \tag{1.55}$$

$$\lambda\phi_{y,j} = \sum_{i:Y^{(i)}=y} x_{i,j} \tag{1.56}$$

$$\phi_{y,j} \propto \sum_{i:Y^{(i)}=y} x_{i,j} = \sum_i \delta(Y^{(i)} = y)x_{i,j}, \tag{1.57}$$

where I use two different notations for indicating the same thing: a sum over the word counts for all documents i such that the label $Y^{(i)} = y$. This gives a solution for each $\boldsymbol{\phi}_y$ up to a constant of proportionality. Now recall the constraint $\forall y, \sum_{j=1}^{V} \boldsymbol{\phi}_{y,j} = 1$; this constraint arises because $\boldsymbol{\phi}_y$ represents a vector of probabilities for each word in the vocabulary. We can exploit this constraint to obtain an exact solution,

$$\phi_{y,j} = \frac{\sum_{i:Y^{(i)}=y} x_{i,j}}{\sum_{j'=1}^{V} \sum_{i:Y^{(i)}=y} x_{i,j'}} \tag{1.58}$$

$$= \frac{\text{count}(y,j)}{\sum_{j'=1}^{V} \text{count}(y,j')}. \tag{1.59}$$

This is exactly equal to the relative frequency estimator. A similar derivation gives $\mu_y \propto \sum_i \delta(Y^{(i)} = y)$, where $\delta(Y^{(i)} = y) = 1$ if $Y^{(i)} = y$ and 0 otherwise.

Smoothing and MAP estimation

If data is sparse, you may end up with values of $\phi = 0$. For example, the word *Bayesian* may have never appeared in a spam email yet, so the relative frequency estimate $\phi_{\text{SPAM},\textit{Bayesian}} = 0$. But choosing a value of 0 would allow this single feature to completely veto a label, since $\Pr(Y = \text{SPAM} \mid \boldsymbol{x}) = 0$ if $\boldsymbol{x}_{\textit{Bayesian}} > 0$.

This is undesirable, because it imposes high **variance**: depending on what data happens to be in the training set, we could get vastly different classification rules. One solution is to **smooth** the probabilities, by adding "pseudo-counts" of α to each count, and then normalizing.

$$\phi_{y,j} = \frac{\alpha + \sum_{i:Y^{(i)}=y} x_j^{(i)}}{\sum_{j'=1}^{V} \left(\alpha + \sum_{i:Y^{(i)}=y} x_{i,j'}\right)} = \frac{\alpha + \text{count}(y,j)}{V\alpha + \sum_{j'=1}^{V} \text{count}(y,j')} \tag{1.60}$$

This form of smoothing is called "Laplace smoothing", and it has a nice Bayesian justification, in which we extend the generative story to include $\boldsymbol{\phi}$ as a random variable (rather than as a parameter). The resulting estimate is called *maximum a posteriori*, or MAP.

Smoothing reduces **variance**, but it takes us away from the maximum likelihood estimate: it imposes a **bias**. In this case, the bias points towards uniform probabilities. Machine learning theory shows that errors on heldout data can be attributed to the sum of bias and variance. Techniques for reducing variance typically increase the bias, so there is a **bias-variance tradeoff**.[8]

- Unbiased classifiers **overfit** the training data, yielding poor performance on unseen data.
- But if we set a very large smoothing value, we can **underfit** instead. In the limit of $\alpha \rightarrow \infty$, we have zero variance: it is the same classifier no matter what data we see! But the bias of such a classifier will be high.
- Navigating this tradeoff is hard. But in general, as you have more data, variance is less of a problem, so you just go for low bias.
- You may wonder if it is possible to choose a separate α_j for each word j, possibly to add larger amounts of smoothing to more common words. Indeed this is possible, and we will talk a great deal about more advanced smoothing techniques in Chapter 5. But I am unaware of any cases where this makes a major positive impact on classification.

Training, testing, and tuning (development) sets

We'll soon talk about more learning algorithms, but whichever one we apply, we will want to report its accuracy. Really, this is an educated guess about how well the algorithm will do on new data in the future.

To make an estimate of the accuracy, we need to hold out a separate "test set" from the data that we use for estimation (i.e., training, learning). Otherwise, if we measure accuracy on the same data that is used for estimation, we will badly overestimate the accuracy that we are likely to get on new data.

Recall that in addition to the parameters $\boldsymbol{\mu}$ and $\boldsymbol{\phi}$, which are learned on training data, we also have the amount of smoothing, α. This can be considered a "tuning" parameter, and it controls the tradeoff between overfitting and underfitting the training data. Where is the best position on this tradeoff curve? It's hard to tell in advance. Sometimes it is tempting to see which tuning parameter gives the best performance on the test set, and then report that performance. Resist this temptation! It will also lead to overestimating accuracy on truly unseen future data. For that reason, this is a sure way to get your research paper rejected; in a commercial setting, this mistake may cause you to promise much higher accuracy than you can deliver. Instead, you should split off a piece of your training data, called a "development set" (or "tuning set").

[8] The bias-variance tradeoff is covered by Murphy (2012), but see Mohri et al. (2012) for a more formal treatment of this key concept in machine learning theory.

Sometimes, people average across multiple test sets and/or multiple development sets. One way to do this is to divide your data into "folds," and allow each fold to be the development set one time. This is called **K-fold cross-validation**. In the extreme, each fold is a single data point. This is called **leave-one-out**.

The Naïvety of Naïve Bayes

Naïve Bayes is simple to work with: estimation and prediction can done in closed form, and the nice probabilistic interpretation makes it relatively easy to extend the model in various ways. But Naïve Bayes makes assumptions which seriously limit its accuracy, especially in NLP.

- The multinomial distribution assumes that each word is generated independently of all the others (conditioned on the parameter ϕ_y). Formally, we assume conditional independence:
$$\text{p}(\textit{naïve}, \textit{Bayes} \mid y) = \text{p}(\textit{naïve} \mid y)\text{p}(\textit{Bayes} \mid y). \tag{1.61}$$
- But this is clearly wrong, because words "travel together." To hone your intuitions about this, try and decide whether you believe
$$\text{p}(\textit{naïve Bayes}) > \text{p}(\textit{naïve})\text{p}(\textit{Bayes}) \tag{1.62}$$
or...
$$\text{p}(\textit{naïve Bayes}) < \text{p}(\textit{naïve})\text{p}(\textit{Bayes}). \tag{1.63}$$
Apply the chain rule!

Traffic lights Dan Klein makes this point with an example about traffic lights. In his hometown of Pittsburgh, there is a $1/7$ chance that the lights will be broken, and both lights will be red. There is a $3/7$ chance that the lights will work, and the north-south lights will be green; there is a $3/7$ chance that the lights work and the east-west lights are green.

The *prior* probability that the lights are broken is $1/7$. If they are broken, the conditional likelihood of each light being red is 1. The prior for them not being broken is $6/7$. If they are not broken, the conditional likelihood of each individual light being red is $1/2$.

Now, suppose you see that both lights are red. According to Naïve Bayes, the probability that the lights are broken is $1/7 \times 1 \times 1 = 1/7 = 4/28$. The probability that the lights are not broken is $6/7 \times 1/2 \times 1/2 = 6/28$. So according to naive Bayes, there is a 60% chance that the lights are not broken!

What went wrong? We have made an independence assumption to factor the probability $\text{p}(R, R \mid \text{not-broken}) = \text{p}_{\text{north-south}}(R \mid \text{not-broken})\text{p}_{\text{east-west}}(R \mid \text{not-broken})$. But this independence assumption is clearly incorrect, because $\text{p}(R, R \mid \text{not-broken}) = 0$.

Less Naïve Bayes? Of course we could decide not to make the naive Bayes assumption, and model $\text{p}(R, R)$ explicitly. But this idea does not scale when the feature space is large — as it often is in NLP. The number of possible feature configurations grows exponentially, so our ability to estimate accurate parameters will suffer from high variance. With an infinite amount of data, we would be okay; but we never have that. Naïve Bayes accepts some bias, because of the incorrect modeling assumption, in exchange for lower variance.

Problems

Chapter 2

Discriminative learning

Naïve Bayes is a simple classifier, where both the prediction rule and the learning objective are based on the joint probability of labels and base features,

$$\log \mathrm{p}(y^{(i)}, \boldsymbol{x}^{(i)}) = \log \mathrm{p}(\boldsymbol{x}^{(i)} \mid y^{(i)}) + \log \mathrm{p}(y^{(i)}) \tag{2.1}$$

$$= \sum_j \log \mathrm{p}(x_{i,j} \mid y^{(i)}) + \log \mathrm{p}(y^{(i)}) \tag{2.2}$$

$$= \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.3}$$

Equation 2.2 shows the independence assumption that makes it possible to compute this joint probability: the probability of each base feature $x_{i,j}$ is mutually independent, after conditioning on the label $y^{(i)}$.

In the equations above, we define the **feature function** $\boldsymbol{f}(\boldsymbol{x}, y)$ so that it corresponds to "bag-of-words" features. Bag-of-words features violate the assumption of conditional independence — for example, the probability that a document will contain the word *naïve* is surely higher given that it also contains the word *Bayes* — but this violation is relatively mild. However, to get really good performance on text classification and other language processing tasks, we will need to add many other types of features. Some of these features will capture parts of words, and others will capture multi-word units. For example:

- Prefixes, such as *anti-*, *im-*, and *un-*.
- Punctuation and capitalization.
- **Bigrams** , such as *not good*, *not bad*, *least terrible*, and higher-order **n-grams** .

Many of these "rich" features violate the Naïve Bayes independence assumption more severely. Consider what happens if we add feature capturing the word prefix. Under the Naïve Bayes assumption, we make the following approximation,:

$$Pr(\text{word} = \textit{impossible}, \text{prefix} = \textit{im-} \mid y) \approx Pr(\text{prefix} = \textit{im-} \mid y) \times Pr(\text{word} = \textit{impossible} \mid y). \tag{2.4}$$

To test the quality of the approximation, we can manipulate the original probability by applying the chain rule,

$$Pr(\text{word} = \textit{impossible}, \text{prefix} = \textit{im-} \mid y) = Pr(\text{prefix} = \textit{im-} \mid \text{word} = \textit{impossible}, y) \times Pr(\text{word} = \textit{impossible} \mid y) \quad (2.5)$$

But $Pr(\text{prefix} = \textit{im-} \mid \text{word} = \textit{impossible}, y) = 1$, since *im-* is guaranteed to be the prefix for the word *impossible*. Therefore,

$$\begin{aligned} &Pr(\text{word} = \textit{impossible}, \text{prefix} = \textit{im-} \mid y) && (2.6)\\ &\quad = 1 \times Pr(\text{word} = \textit{impossible} \mid y) \\ &\quad \gg Pr(\text{prefix} = \textit{im-} \mid y) \times Pr(\text{word} = \textit{impossible} \mid y). && (2.7) \end{aligned}$$

The final inequality is due to the fact that the probability of any given word starting with the prefix *im-* is much less than one, and it shows that Naïve Bayes will systematically underestimate the true probabilities of conjunctions of positively correlated features. To use such features, we will need learning algorithms that do not rely on an independence assumption.

2.1 Perceptron

In Naïve Bayes, the weights can be interpreted as parameters of a probabilistic model. But this model requires an independence assumption that usually does not hold, and limits our choice of features. Why not forget about probability and learn the weights in an error-driven way? The perceptron algorithm, shown in Algorithm 1, is one way to do this.[1]

What the algorithm says is this: if you make a mistake, increase the weights for features which are active with the correct label $y^{(i)}$, and decrease the weights for features which are active with the guessed label $\hat{y}$. This is an **online learning** algorithm, since the classifier weights change after every example. This is different from Naïve Bayes, which computes corpus statistics and then sets the weights in a single operation — Naïve Bayes is a **batch learning** algorithm.[2]

The perceptron algorithm may seem like a cheap heuristic: Naïve Bayes has a solid foundation in probability, but now we are just adding and subtracting constants from the weights every time there is a mistake. Will this really work? In fact, there is some

[1]The attentive reader will note that Algorithm 1 does not define the initial values of $\boldsymbol{\theta}$ or the index t. Initialization decisions are typically heuristic, and I prefer not to clutter the algorithm definition by committing to one initialization procedure or another. In this case, $\boldsymbol{\theta} = \mathbf{0}$ is a perfectly good choice. I have been similarly vague about the stopping criterion, but the text presents some alternatives. Counters like t should be assumed to begin at $t \leftarrow 1$ unless otherwise noted.

[2]Later in this chapter we will encounter a third class of learning algorithm, which is **iterative**. Such algorithms perform multiple updates to the weights (like perceptron), but are also **batch**, in that they have to use all the training data to compute the update.

Algorithm 1 Perceptron learning algorithm

```
procedure PERCEPTRON(x^(1:N), y^(1:N))
    repeat
        Select an instance i
        ŷ ← argmax_y θ_t^⊤ f(x^(i), y)
        if ŷ ≠ y^(i) then
            θ_{t+1} ← θ_t + f(x^(i), y^(i)) − f(x^(i), ŷ)
        else
            do nothing
    until tired
```

nice theory for the perceptron. To understand it, we must introduce the notion of **linear separability** :

Definition 1 (Linear separability)**.** *The dataset $\mathcal{D} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_i$ is linearly separable iff there exists some weight vector $\boldsymbol{\theta}$ and some **margin** ρ such that for every instance $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$, the inner product of $\boldsymbol{\theta}$ and the feature function for the true label, $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y^{(i)})$, is at least ρ greater than inner product of $\boldsymbol{\theta}$ and the feature function for every other possible label, $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y')$.*

$$\exists \boldsymbol{\theta}, \rho > 0 : \forall \langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle \in \mathcal{D}, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \geq \rho + \max_{y' \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'). \tag{2.8}$$

Linear separability is important because of the following guarantee: if your data is linearly separable, then the perceptron algorithm will find a separator (Novikoff, 1962).[3] So while the perceptron may seem heuristic, it is guaranteed to succeed — if the learning problem is easy enough.

How useful is this proof? Minsky and Papert (1969) note that the simple logical function of *exclusive-or* is not separable, and that a perceptron is therefore incapable of learning to mimic this function. But this is not just a problem for perceptron: any linear classification algorithm, including Naïve Bayes, will fail to learn this function. In natural language, we work in very high dimensional feature spaces, with thousands or millions of features. In these high-dimensional spaces, finding a separator becomes exponentially easier. Furthermore, later theoretical work showed that if the data is not separable, it is still possible to place an upper bound on the number of errors that the perceptron algorithm will make (Freund and Schapire, 1999).

[3] It is also possible to prove an upper bound on the number of training iterations required to find the separator. Proofs like this are part of the field of **statistical learning theory**. Mohri et al. (2012) provide an excellent survey.

Averaged perceptron

The perceptron iterates over the data repeatedly — until "tired", as described in Algorithm 1. If the data is linearly separable, it is guaranteed that the perceptron will eventually find a separator, and then we can stop. But if the data is not separable, the algorithm can *thrash* between two or more weight settings, never converging. In this case, how do we know that we can stop training, and how should we choose the final weights? An effective practical solution is to *average* the perceptron weights across all iterations.

This procedure is shown in Algorithm 2. The learning algorithm is nearly identical to the "vanilla" perceptron, but we also maintain a vector of the weight sums, $\boldsymbol{m}$. At the end of the learning procedure, we divide this sum by the total number of updates t, to compute the averaged weights, $\overline{\boldsymbol{\theta}}$. These averaged weights are then used to predict the labels of new data, such as examples in the test set. Even if the data is not separable, the averaged weights will eventually converge. One possible stopping criterion is to check the difference between the average weight vectors after each pass through the data: if the norm of the difference falls below some predefined threshold, we can stop iterating. Another stopping criterion is to hold out some data, and to measure the predictive accuracy on this heldout data (this is called a *development set* in chapter 1). When the accuracy on the heldout data starts to decrease, the learning algorithm has begun to **overfit** . At this point, it is probably best to stop; this stopping criterion is known as **early stopping** .

Algorithm 2 Averaged perceptron learning algorithm

```
1:  procedure AVG-PERCEPTRON(x^(1:N), y^(1:N))
2:      repeat
3:          Select an instance i
4:          ŷ ← argmax_y θ_t^⊤ f(x^(i), y)
5:          if ŷ ≠ y^(i) then
6:              θ_{t+1} ← θ_t + f(x^(i), y^(i)) − f(x^(i), ŷ)
7:              m ← m + θ_{t+1}
8:          else
9:              do nothing
10:     until tired
11:     θ̄ ← (1/t) m
```

Generalization is the ability to make good predictions on instances that are not in the training data; it can be proved that averaging improves generalization, by computing an upper bound on the generalization error (Freund and Schapire, 1999; Collins, 2002).

2.2 Loss functions and large margin classification

Naïve Bayes chooses the weights $\boldsymbol{\theta}$ by maximizing the joint likelihood $\text{p}(\{\boldsymbol{x}^{(i)}, y^{(i)}\}_i)$. This is equivalent to maximizing the log-likelihood (due to the monotonicity of the log function), and also to **minimizing** the negative log-likelihood. This negative log-likelihood can therefore be viewed as a **loss function** ,

$$\log \text{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\theta}) = \sum_{i=1}^{N} \log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{2.9}$$

$$\ell_{\text{NB}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = -\log \text{p}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}) \tag{2.10}$$

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{argmin}} \sum_{i=1}^{N} \ell_{\text{NB}}(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.11}$$

This minimization problem is identical to the maximum-likelihood estimation problem that we solved in the previous chapter. Framing it as minimization may seem confusing and backwards, but loss functions provide a very general framework in which to compare many approaches to machine learning. For example, even though the perceptron is not a probabilistic model, it is also trying to minimize a **loss function**:

$$\ell_{\text{perceptron}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & y^{(i)} = \text{argmax}_y \, \boldsymbol{\theta}^\top \boldsymbol{f}(x_i, y) \\ 1, & \text{otherwise} \end{cases} \tag{2.12}$$

The perceptron loss — sometimes called the $0/1$ loss — has some pros and cons in comparison with the joint likelihood loss implied by Naive Bayes.

- ℓ_{NB} can suffer **infinite** loss on a single example, which suggests it will overemphasize some examples, and underemphasize others.
- $\ell_{\text{perceptron}}$ treats all errors equally. It only cares if the example is correct, and not about how confident the classifier was. Since we usually evaluate on accuracy or some related error-based metric, this is a better match.
- $\ell_{\text{perceptron}}$ is non-convex[4] and discontinuous. Although it is possible to bound the number of errors on the training data, finding the **global optimum** is intractable when the data is not separable.

We can fix this last problem by defining a loss function that behaves more nicely. To do this, let's define the **margin** as

$$\gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \max_{y \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{2.13}$$

[4] A function f is convex iff $\alpha f(x_i) + (1-\alpha) f(x_j) \geq f(\alpha x_i + (1-\alpha) x_j)$, for all $\alpha \in [0, 1]$ and for all x_i and x_j on the domain of the function. Convexity implies that any local minimum is also a global minimum, and there are effective techniques for optimizing convex functions (Boyd and Vandenberghe, 2004).

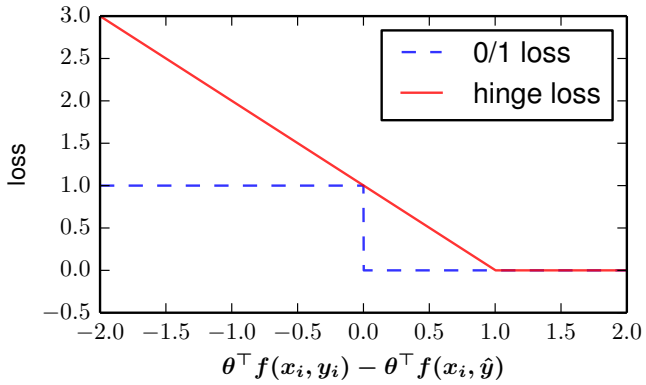

Figure 2.1: Hinge and perceptron loss functions

The margin represents the separation between the score for the correct label $y^{(i)}$, and the score for the highest-scoring label. If the instance is classified incorrectly, the margin will be negative. The intuition behind "large-margin" learning algorithms is that it is not enough just to get the training data correct — we want the correct label to be separated from the other possible labels by a comfortable margin. We can use the margin to define a convex and continuous **hinge loss** ,

$$\ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \begin{cases} 0, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1, \\ 1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), & \text{otherwise} \end{cases} \tag{2.14}$$

Equivalently, we can write $\ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = \left(1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})\right)_+$, where $(x)_+$ is equal to x if x is positive, and 0 otherwise. The hinge loss is zero if we have a margin of at least 1 between the score for the true label and the best-scoring alternative, which we have written $\hat{y}$. The hinge and perceptron loss functions are shown in Figure 2.1. Note that the hinge loss is an upper bound on the perceptron loss.

Support vector machines

We can write the weight vector $\boldsymbol{\theta} = s\boldsymbol{u}$, where the **norm** of $\boldsymbol{u}$ is equal to one, $||\boldsymbol{u}||_2 = 1$.[5] Think of s as the magnitude and $\boldsymbol{u}$ as the direction of the vector $\boldsymbol{\theta}$. If the data is separable, there are many values of s that attain zero hinge loss. To see this, let us redefine the margin

[5]The norm of a vector $||\boldsymbol{u}||_2$ is defined as, $||\boldsymbol{u}||_2 = \sqrt{\sum_j u_j^2}$.

as,

$$\gamma(\boldsymbol{\theta}, \boldsymbol{x}^{(i)}, y^{(i)}) = \min_{y \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \tag{2.15}$$

$$= \min_{y \neq y^{(i)}} s(\boldsymbol{u}^\top \Big(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) \Big) . \tag{2.16}$$

Based on this definition, if the unit vector $\boldsymbol{u}^*$ satisfies $\gamma(\boldsymbol{u}^*, \boldsymbol{x}^{(i)}, y^{(i)}) > 0$, then there is some smallest value s^* such that $\forall s \geq s^*, \gamma(s\boldsymbol{u}^*, \boldsymbol{x}^{(i)}, y^{(i)}) \geq 1$. This observation suggests that given many possible $\boldsymbol{\theta}$ that obtain zero hinge loss, we should choose the one with the smallest norm ($s = s^*$), since this entails making the least committment to the training data. This idea underlies the **Support Vector Machine** (SVM) classifier, which, in its most basic form, solves the following optimization problem,

$$\begin{aligned} \min_{\boldsymbol{\theta}} . \quad & ||\boldsymbol{\theta}||_2^2 \\ s.t. \quad & \forall_i \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) = 0. \end{aligned} \tag{2.17}$$

Recall that $||\boldsymbol{\theta}||_2^2 = \sum_j \theta_j^2$.

In realistic settings, we do not know whether there is any feasible solution — that is, whether there exists any $\boldsymbol{\theta}$ so that the hinge loss on every training instance is zero. We therefore introduce a set of **slack variables** $\xi_i \geq 0$, which represent a sort of "fudge factor" for each instance i — instead of requiring that the hinge loss be exactly zero, we require that it be less than ξ_i. Ideally there would not be any slack, so we add the sum of the slack variables to the objective function to be minimized:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & ||\boldsymbol{\theta}||_2^2 + C \sum_i \xi_i \\ s.t. \quad & \forall_i \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \leq \xi_i \\ & \forall_i \xi_i \geq 0. \end{aligned} \tag{2.18}$$

Here C is a tunable parameter that controls the penalty on the slack variables. As $C \rightarrow \infty$, slack is infinitely expensive, and we can only find a solution if the data is separable. As $C \rightarrow 0$, slack becomes free, and there is a trivial solution at $\boldsymbol{\theta} = \mathbf{0}$, regardless of the data. Thus, C plays a similar role to the smoothing parameter in Naïve Bayes (section 1.2), trading off between a close fit to the training data and better generalization. Like the smoothing parameter of Naïve Bayes, C must be set by the user, typically by maximizing performance on a heldout development set.

To solve the constrained optimization problem defined in Equation 2.18, we can use Lagrange multipliers to convert it into the unconstrained **primal form**,[6]

$$\min_{\boldsymbol{\theta}} \quad \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}), \tag{2.19}$$

where λ is a tunable parameter that can be computed from the term C in Equation 2.18. A generic way to minimize such objective functions is **gradient descent** : moving along the gradient (obtained by differentiating with respect to $\boldsymbol{\theta}$), until the gradient is equal to zero.[7]

Let us rewrite the primal form of the SVM optimization problem as follows:

$$L_{SVM} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.20}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}))_+ \tag{2.21}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 + \sum_i^N (1 - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \max_{y \neq y^{(i)}} \boldsymbol{f}(\boldsymbol{x}^{(i)}, y))_+ \tag{2.22}$$

Then the (sub)gradient of Equation 2.22 is:

$$\frac{\partial L_{SVM}}{\partial \boldsymbol{\theta}} = \lambda \boldsymbol{\theta} + \sum_i^N \delta_i (\max_{y \neq y^{(i)}} \boldsymbol{f}(\boldsymbol{x}^{(i)}, y) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})) \tag{2.23}$$

$$\delta_i \triangleq \begin{cases} 1, & \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) < 1 \\ 0, & \text{otherwise.} \end{cases} \tag{2.24}$$

The term δ_i can be thought of as a gate: when the margin is less than one, the gradient includes the difference in features between the true label and the next best predicted label; when the margin is more than one, the gradient does not include this term. Because L_{SVM} is a convex function of $\boldsymbol{\theta}$, this gradient is equal to zero only at the global minimum. Gradient-based optimization techniques are discussed in section 2.4.

[6] An alternative **dual form** is used in the formulation of the kernel-based support vector machine, which supports non-linear classification. This is described briefly at the end of the chapter.

[7] Because the hinge loss is not smooth, there is not a single gradient at the point at which the hinge loss is exactly equal to zero, but rather, a **subgradient set** . However, this is a theoretical issue that poses no difficulties in practice.

2.3 Logistic regression

Thus far, we have seen two broad classes of learning algorithms. Naïve Bayes is a probabilistic method, where learning is equivalent to estimating a joint probability distribution. Perceptron, support-vector machines (SVM), and passive-aggressive (PA) are all error-driven algorithms: the learning objective is to minimize the number of errors on the training data (perceptron), or to minimize a convex upper bound on the number of errors (SVM, PA). Both approaches have advantages: probability enables us to quantify uncertainty about the predicted labels, but error-driven learning typically leads to better performance on error-based performance metrics such as accuracy.

Logistic regression combines both of these advantages: it is error-driven like the perceptron and margin-based learning algorithms, but it is probabilistic like Naïve Bayes. To understand the motivation for logistic regression, first recall that Naïve Bayes selects weights to optimize the joint probability $\text{p}(\boldsymbol{x}, y)$.

- We have used the chain rule to factor this joint probability as $\text{p}(\boldsymbol{x}, y) = \text{p}(\boldsymbol{x} \mid y) \times \text{p}(y)$.
- But we could equivalently choose the alternative factorization $\text{p}(\boldsymbol{x}, y) = \text{p}(y \mid \boldsymbol{x}) \times \text{p}(\boldsymbol{x})$.

In classification, we always know $\boldsymbol{x}$: these are the base features from which we predict y. So there is no need to model $\text{p}(\boldsymbol{x})$; we really care only about the **conditional probability** $\text{p}(y \mid \boldsymbol{x})$ — sometimes called the **likelihood** . Logistic regression defines this probability directly, in terms of the features $\boldsymbol{f}(\boldsymbol{x}, y)$ and the weights $\boldsymbol{\theta}$.

We can think of $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$ as a scoring function for the compatibility of the base features $\boldsymbol{x}$ and the label y. This function is an unconstrained scalar; we would like to convert it to a probability. To do this, we first **exponentiate**, obtaining $\exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)\right)$, which is guaranteed to be non-negative. Next, we need to **normalize**, dividing over all possible labels $y' \in \mathcal{Y}$. The resulting conditional probability is defined as,

$$\text{p}(y \mid \boldsymbol{x}) = \frac{\exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y')\right)}. \tag{2.25}$$

Given a dataset $\mathcal{D} = \{\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle\}_i$, the maximum-likelihood estimator for $\boldsymbol{\theta}$ is obtained

by maximizing,

$$L(\boldsymbol{\theta}) = \log \mathrm{p}(\boldsymbol{y}^{(1:N)} \mid \boldsymbol{x}^{(1:N)}; \boldsymbol{\theta}) \tag{2.26}$$
$$= \log \prod_i \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{2.27}$$
$$= \sum_i \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{2.28}$$
$$= \sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right). \tag{2.29}$$

The final line is obtained by plugging in Equation 2.25 and taking the logarithm.[8,9] Inside the sum, we have the (additive inverse of the) **logistic loss**.

- In binary classification, we can write this as

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -(y^{(i)} \boldsymbol{\theta}^\top \boldsymbol{x}_i - \log\left(1 + \exp \boldsymbol{\theta}^\top \boldsymbol{x}_i\right)) \tag{2.30}$$

- In multi-class classification, we have,

$$\ell_{\text{logistic}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')) \tag{2.31}$$

The logistic loss is shown in Figure 2.2. Note that logistic loss is also an upper bound on the perceptron loss. A key difference from the perceptron and hinge losses is that logistic loss is *never* exactly zero: the objective function can always be improved by chosing the correct label with more confidence.

Regularization

As with the margin-based algorithms described in section 2.2, we can obtain better generalization by penalizing the norm of $\boldsymbol{\theta}$, by adding a term of $\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$ to the minimization objective. This is called L_2 regularization, because it includes the L_2 norm. It can be viewed as placing a zero-mean Gaussian prior distribution on each term of $\boldsymbol{\theta}$, because the log-likelihood under a zero-mean Gaussian is,

$$\log N(\theta_j; 0, \sigma^2) \propto -\frac{1}{2\sigma^2}\theta_j^2, \tag{2.32}$$

[8] Any reasonable base will work; if it is important to you to know which one to choose, then I suggest using base 2 if you are a computer scientist, and base e otherwise.

[9] The log-sum-exp term is very common in machine learning. It is numerically unstable because it will underflow if the inner product is small, and overflow if the inner product is large. Scientific computing libraries usually contain special functions for computing `logsumexp`, but with some thought, you should be able to see how to create an implementation that is numerically stable.

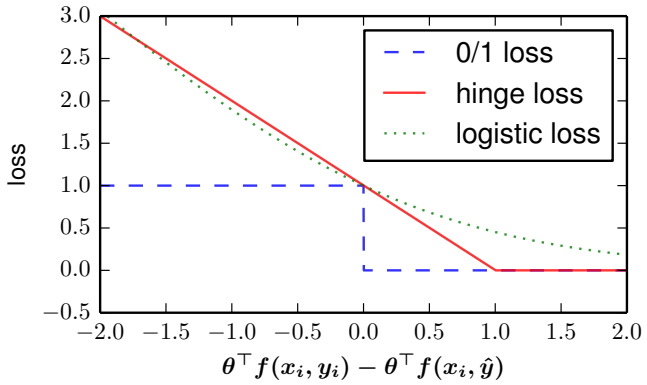

Figure 2.2: Hinge, perceptron, and logistic loss functions

so that $\lambda = \frac{1}{\sigma^2}$.

The effect of this regularizer will cause the estimator to trade off conditional likelihood on the training data for a smaller norm of the weights, and this can help to prevent overfitting. Indeed, regularization is generally considered to be essential to estimating high-dimensional models, as we typically do in NLP. To see why, consider what would happen to the unregularized weight for a base feature j that was active in only one instance $\boldsymbol{x}^{(i)}$: the conditional likelihood could always be improved by increasing the weight for this feature, so that $\boldsymbol{\theta}_{(j,y^{(i)})} \rightarrow \infty$ and $\boldsymbol{\theta}_{(j,\tilde{y} \neq y^{(i)})} \rightarrow -\infty$, where (j, y) indicates the index of feature associated with $x_{i,j}$ and label y in $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)$.

Gradients

We will optimize $\boldsymbol{\theta}$ through gradient descent. Specific algorithms are described in section 2.4, but because the gradient of the logistic regression objective is illustrative, it is

worth working out in detail. Let us begin with the logistic loss on a single example,

$$\ell(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) = -(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \tag{2.33}$$

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \frac{1}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \sum_{y'} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.34}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y'} \frac{\exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right)}{\sum_{y'' \in \mathcal{Y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'')\right)} \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.35}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y'} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \times \boldsymbol{f}(\boldsymbol{x}^{(i)}, y') \tag{2.36}$$

$$= -\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)], \tag{2.37}$$

where the final step employs the definition of an expectation (section 1.1). The gradient thus has the pleasing interpretation as the difference between the observed feature counts $\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)})$ and the expected counts under the current model, $E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$. When these two count vectors are equal for a single example, there is nothing more to learn from this example; when they are equal in sum over the entire dataset, there is nothing more to learn from the dataset as a whole.

As we will see shortly, a simple online approach to gradient-based optimization is to take a step along the gradient. In (unregularized) logistic regression, this gradient-based optimization is a soft version of the perceptron. Put another way, in the case that $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{x})$ is a delta function, $\mathrm{p}(y \mid \boldsymbol{x}) = \delta(y = \hat{y})$, then the gradient step is exactly equal to the perceptron update.

If we add a regularizer $\frac{\lambda}{2}||\boldsymbol{\theta}||_2^2$, then this contributes $\lambda\boldsymbol{\theta}$ to the overall gradient:

$$L = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_{i=1}^{N} \left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_{y' \in \mathcal{Y}} \exp \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y')\right) \tag{2.38}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}} = \lambda\boldsymbol{\theta} - \sum_{i=1}^{N} \left(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]\right) \tag{2.39}$$

2.4 Optimization

In Naïve Bayes, the gradient on the joint likelihood led us to a closed form solution for the parameters $\boldsymbol{\theta}$; in passive-aggressive, we obtained a solution for each individual update from a constrained optimization problem. In logistic regression and support vector machines (SVM), we have objective functions L.

- In logistic regression, L corresponds to the regularized negative log-likelihood,

$$L_{LR} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_i \left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \log \sum_y \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y)\right) \right) \tag{2.40}$$

- In the support vector machine, L corresponds to the "primal form",

$$L_{SVM} = \frac{\lambda}{2}||\boldsymbol{\theta}||_2^2 - \sum_i (1 - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \max_{y' \neq y^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'))_+ \tag{2.41}$$

In both cases, the objective is convex, and there are many efficient algorithms for optimizing convex functions (Boyd and Vandenberghe, 2004). Most algorithms are based on the **gradient** $\frac{\partial L}{\partial \boldsymbol{\theta}}$, or on the subgradients, in the case of non-smooth objectives in which the gradient is not unique. This section will present the most frequently-used optimization algorithms, focusing on logistic regression. However, these algorithms can also be applied to the support vector machine objective with minimal modification.

Batch optimization

In batch optimization, all the data is kept in memory and iterated over many times. The logistic loss is smooth and convex, so we can find the global optimum using gradient descent,

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta_t \frac{\partial L}{\partial \boldsymbol{\theta}}, \tag{2.42}$$

where $\frac{\partial L}{\partial \boldsymbol{\theta}}$ is the gradient computed over the entire training set, and η_t is some **step size**. In practice, this can be very slow to converge, as the gradient can become infinitesimally small. Second-order (Newton) optimization obtains much better convergence rates by incorporating the inverse of the Hessian matrix,

$$H_{i,j} = \frac{\partial^2}{\partial w_i \partial w_j} L. \tag{2.43}$$

Unfortunately, in NLP problems, the Hessian matrix (which is quadratic in the number of parameters) is usually too big to deal with. A typical solution is to approximate the Hessian matrix via a **quasi-Newton optimization** technique, such as L-BFGS (Liu and Nocedal, 1989).[10] Quasi-Newton optimization packages are available in many scientific computing environments, and for most types of NLP practice and research, it is okay to treat them as black boxes. You will typically pass in a pointer to a function that computes the likelihood and gradient, and the solver will return a set of weights.

[10] You can remember the order of the letters as "Large Big Friendly Giants." Does this help you?

Online optimization

In online optimization, you consider one example (or a "mini-batch" of a few examples) at a time. **Stochastic gradient descent** (SGD) makes a stochastic online approximation to the overall gradient. Here is the SGD update for logistic regression:

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta_t \frac{\partial L_{LR}}{\partial \boldsymbol{\theta}} \tag{2.44}$$

$$= \boldsymbol{\theta}^{(t)} - \eta_t \left(\lambda \boldsymbol{\theta}^{(t)} - \sum_i^N \left(\boldsymbol{f}(\boldsymbol{x}_i, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_i, y)] \right) \right) \tag{2.45}$$

$$= (1 - \lambda\eta_t)\boldsymbol{\theta}^{(t)} + \eta_t \left(\sum_i^N \boldsymbol{f}(\boldsymbol{x}_i, y^{(i)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_i, y)] \right) \tag{2.46}$$

$$\approx (1 - \lambda\eta_t)\boldsymbol{\theta}^{(t)} + N\eta_t \left(\boldsymbol{f}(\boldsymbol{x}_{i(t)}, y_{i(t)}) - E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}_{i(t)}, y)] \right) \tag{2.47}$$

where η_t is the **step size** at iteration t, and $\langle \boldsymbol{x}_{i(t)}, y_{i(t)} \rangle$ is an instance that is *randomly sampled* at iteration t. We can obtain a more compact form for SGD by folding the constant N into η_t and λ, so that $\tilde{\eta}_t = N\eta_t$ and $\tilde{\lambda} = \frac{\lambda}{N}$. This yields the form shown in Algorithm 3. A similar online algorithm can be derived for the SVM objective, using the subgradient in Equation 2.23.

Algorithm 3 Stochastic gradient descent for logistic regression

```
procedure SGD(x^(1:N), y^(1:N), η, λ)
    t ← 1
    repeat
        Select an instance i
        θ^(t+1) ← (1 − λ̃η̃_t)θ^(t) + η̃_t (f(x^(i), y^(i)) − E_{y|x}[f(x^(i), y^(i))])
        t ← t + 1
    until tired
```

As above, the expectation is equal to a weighted sum over the labels,

$$E_{y|\boldsymbol{x}}[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)] = \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \boldsymbol{f}(\boldsymbol{x}^{(i)}, y'). \tag{2.48}$$

Again, note how similar this update is to the perceptron.

The theoretical foundation for SGD assumes that each training instance is randomly sampled (thus the name "stochastic"), but in practice, it is typical to stream through the data sequentially. It is often useful to select not a single instance, but a **mini-batch** of K instances. In this case, we would scale η_t and λ by $\frac{N}{K}$. The gradients over mini-batches will be lower variance approximations of the true gradient, and it is possible to parallelize the computation of the gradient for each instance in the mini-batch.

A key question for SGD is how to set the learning rates η_t. It can be proven that SGD will converge if $\eta_t = \eta_0 t^{-\alpha}$ for $\alpha \in [1, 2]$; however, convergence may be very slow. In practice, η_t may also be fixed to a small constant, like 10^-3. In either case, it is typical to try a set of different values, and see which minimizes the objective L most quickly. For more on stochastic gradient descent, as applied to a number of different learning algorithms, see (Zhang, 2004) and (Bottou, 1998). Murphy (2012) traces SGD to Nemirovski and Yudin (1978).

AdaGrad

There are a number of ways to improve on stochastic gradient descent (Bottou et al., 2016). For NLP applications, a popular choice is use an **adaptive** step size, which can be different for every feature (Duchi et al., 2011). Features that occur frequently are likely to be updated frequently, so it is best to use a small step size; rare features will be updated infrequently, so it is better to take larger steps. The **AdaGrad** (adaptive gradient) algorithm achieves this behavior by storing the sum of the squares of the gradients for each feature, and rescaling the learning rate by its inverse:

$$\boldsymbol{g}_t = \lambda\boldsymbol{\theta} - \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) + \sum_{y' \in \mathcal{Y}} \mathrm{p}(y' \mid \boldsymbol{x}^{(i)}) \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) \tag{2.49}$$

$$\theta_j^{(t+1)} \leftarrow \theta_j^{(t)} - \frac{\eta}{\sqrt{\sum_{t'=1}^{t} g_{t,j}^2}} g_{t,j}, \tag{2.50}$$

where j iterates over features in $\boldsymbol{f}(\boldsymbol{x}, y)$. AdaGrad seems to require less careful tuning of η, and Dyer (2014) reports that $\eta = 1$ works for a wide range of problems.

2.5 Additional topics in classification*

Passive-aggressive

In online learning, rather than seeking the feasible $\boldsymbol{\theta}$ with the smallest norm, we might instead prefer to make the smallest magnitude **change** to $\boldsymbol{\theta}$, while meeting the hinge loss constraint for instance $\langle \boldsymbol{x}^{(i)}, y^{(i)} \rangle$. Specifically, at each step t, we solve the following optimization problem:

$$\begin{aligned} \min w. \quad & \frac{1}{2}||\boldsymbol{\theta} - \boldsymbol{\theta}_t||^2 + C\xi_t \\ s.t. \quad & \ell_{\text{hinge}}(\boldsymbol{\theta}; \boldsymbol{x}_i, y^{(i)}) \leq \xi_t, \xi_t \geq 0 \end{aligned} \tag{2.51}$$

By forming another Lagrangian, it is possible to show that the solution to Equation 2.51 is,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \tau_t(\boldsymbol{f}(y^{(i)}, \boldsymbol{x}^{(i)}) - \boldsymbol{f}(\hat{y}, \boldsymbol{x}^{(i)})) \tag{2.52}$$

$$\tau_t = \min\left(C, \frac{\ell(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})}{||\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})||^2}\right), \tag{2.53}$$

This algorithm is called **Passive-Aggressive** (PA; Crammer et al., 2006), because it is passive when the margin constraint is satisfied, but it aggressively changes the weights to satisfy the constraints if necessary.[11] PA is error-driven like the perceptron, and the update is nearly identical: the only difference is the learning rate τ_t, which depends on the amount of loss incurred by instance i, the norm of the difference in feature vectors between the predicted and correct labels, and the hyperparameter C, which places an upper bound on the step size. As with the perceptron, it is possible to apply weight averaging to PA, which can improve generalization. PA allows more explicit control than the Averaged Perceptron, due to the C parameter: when C is small, we make very conservative adjustments to $\boldsymbol{\theta}$ from each instance, because the slack variables aren't very expensive; when C is large, we make large adjustments to avoid using the slack variables.

Other regularizers

In Equation 2.38, we proposed to **regularize** the logistic regression estimator by penalizing the squared L_2 norm, $||\boldsymbol{\theta}||_2^2$. However, this is not the only way to penalize large weights; we might prefer some other norm, such as $L_0 = ||\boldsymbol{\theta}||_0 = \sum_j \delta(\theta_j \neq 0)$, which applies a constant penalty for each non-zero weight. This norm can be thought of as a form of **feature selection**: optimizing the L_0-regularized conditional likelihood is equivalent to trading off the log-likelihood against the number of active features. Reducing the number of active features is desirable because the resulting model will be fast, low-memory, and should generalize well, since features that are not very helpful will be pruned away. Unfortunately, the L_0 norm is non-convex and non-differentiable; optimization under L_0 regularization is NP-hard, meaning that it can be solved efficiently only if P=NP (Ge et al., 2011).

A useful alternative is the L_1 norm, which is equal to the sum of the absolute values of the weights, $||\boldsymbol{\theta}||_1 = \sum_j |\theta_j|$. The L_1 norm is convex, and can be used as an approximation to L_0 (Tibshirani, 1996). Moreover, the L_1 norm also performs feature selection, by driving many of the coefficients to zero; it is therefore known as a **sparsity inducing regularizer** . Gao et al. (2007) compare L_1 and L_2 regularization on a suite of NLP problems, finding that L_1 regularization generally gives similar test set accuracy to L_2 regularization,

[11] A related algorithm without slack variables is called MIRA, for Margin-Infused Relaxed Algorithm (Crammer and Singer, 2003).

but that L_1 regularization produces models that are between ten and fifty times smaller, because more than 90% of the feature weights are set to zero.

The L_1 norm does not have a gradient at $\theta_j = 0$, so we must instead optimize the L_1-regularized objective using **subgradient** methods. The associated stochastic subgradient descent algorithms are only somewhat more complex than conventional SGD; Sra et al. (2012) survey approaches for estimation under L_1 and other regularizers.

Other views of logistic regression

Logistic regression is so named because in the binary case where $y \in \{0, 1\}$, we are performing a regression of $\boldsymbol{x}$ against $\boldsymbol{y}$, after passing the inner product $\boldsymbol{\theta}^\top \boldsymbol{x}$ through a logistic transformation to obtain a probability. However, it goes by many other names:

- Logistic regression is also called **maximum conditional likelihood** (MCL), because it is based on maximizing the conditional likelihood $\mathrm{p}(y \mid \boldsymbol{x})$.
- Logistic regression can be viewed as part of a larger family of **generalized linear models** (GLMs), which include other "link functions," such as the probit function. If you use the R software environment, you may be familiar with `glmnet`, a widely-used package for estimating GLMs.
- In the neural networks literature, the multivariate analogue of the logistic transformation is sometimes called a **softmax** layer, because it "softly" identifies the label y that maximizes the activation function $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$.

In the early NLP literature, logistic regression is frequently called **maximum entropy** (Berger et al., 1996). This is due to an alternative formulation, which tries to find the maximum entropy probability function that satisfies moment-matching constraints. The moment matching constraints specify that the empirical counts of each label-feature pair should match the expected counts:

$$\forall j, \sum_{i=1}^{N} f_j(\boldsymbol{x}^{(i)}, y^{(i)}) = \sum_{i=1}^{N} \sum_{y \in \mathcal{Y}} \mathrm{p}(y \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{x}^{(i)}, y) \tag{2.54}$$

Note that this constraint will be met exactly when the derivative of the likelihood function (Equation 2.37) is equal to zero. However, this constraint can be met for many values of $\boldsymbol{\theta}$, so which should we choose?

The **entropy** of the conditional likelihood $\mathrm{p}_{y|\boldsymbol{x}}$ is,

$$H(\mathrm{p}_{y|\boldsymbol{x}}) = -\sum_{\boldsymbol{x} \in \mathcal{X}} \mathrm{p}_{\boldsymbol{x}}(\boldsymbol{x}) \sum_{y \in \mathcal{Y}} \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}) \log \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}), \tag{2.55}$$

where $\mathrm{p}_{\boldsymbol{x}}(\boldsymbol{x})$ is the probability of observing the base features $\boldsymbol{x}$. We compute an empirical estimate of the entropy by summing over all the instances in the training set,

$$\tilde{H}(\mathrm{p}_{y|\boldsymbol{x}}) = -\frac{1}{N}\sum_{i}\sum_{y\in\mathcal{Y}} \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}^{(i)}) \log \mathrm{p}_{y|\boldsymbol{x}}(y \mid \boldsymbol{x}^{(i)}). \tag{2.56}$$

If the entropy is large, the likelihood function is smooth across possible values of y; if it is small, the likelihood function is sharply peaked at some preferred value; in the limiting case, the entropy is zero if $\mathrm{p}(y \mid x) = 1$ for some y. By saying we want a maximum-entropy classifier, we are saying we want to make the weakest commitments possible, while satisfying the moment-matching constraints from Equation 2.54. The solution to this constrained optimization problem is identical to the maximum conditional likelihood (logistic-loss) formulation we considered in the previous section. This view of logistic regression is arguably a little dated, but it is useful to understand, especially when reading classic papers from the 1990s. For a tutorial on maximum entropy, see `http://www.cs.cmu.edu/afs/cs/user/aberger/www/html/tutorial/tutorial.html`.

2.6 Summary of learning algorithms

Having seen several learning algorithms, it is natural to ask which is best in various situations.

Naïve Bayes *Pros*: easy to implement; estimation is very fast, requiring only a single pass over the data; assigns probabilities to predicted labels; controls overfitting with smoothing parameter. *Cons*: the joint likelihood is arguably the wrong objective to optimize; often has poor accuracy, especially with correlated features.

Perceptron and PA *Pros*: easy to implement; online learning means it is not necessary to store all data in memory; error-driven learning means that accuracy is typically high, especially after averaging. *Cons*: not probabilistic, which can be bad in pipeline architectures, when the output of one system becomes the input for another; non-averaged perceptron performs poorly if data is not separable; hard to know when to stop learning.

Support vector machine *Pros*: optimizes an error-based metric, usually resulting in high accuracy; overfitting is controlled by a regularization parameter. *Cons*: batch learning requires black-box optimization; not probabilistic.

Logistic regression *Pros*: error-driven and probabilistic; overfitting is controlled by a regularization parameter. *Cons*: batch learning requires black-box optimization; logistic loss sometimes gives lower accuracy than hinge loss, due to overtraining on correctly-labeled examples.

Table 2.1 summarizes some properties of Naïve Bayes, perceptron, PA, and logistic regression. SVM is left out because it is identical to PA on most of these dimensions, except for the estimation procedure, which typically employs a black-box convex optimization package. In non-probabilistic settings, I usually reach for averaged perceptron first if I am coding from scratch, and SVM if I am using a library of learning algorithms such as `sklearn`. If probabilities are necessary, I use logistic regression.

What about non-linear classification?

The feature spaces that we consider in NLP are usually huge, so non-linear classification can be quite difficult. When the feature dimension V is larger than the number of instances N — often the case in NLP — you can always learn a linear classifier that will perfectly classify your training instances.[12] This makes selecting an appropriate **non-linear** classifier especially difficult. Nonetheless, there are some approaches to non-linear learning in NLP:

- The most common approach is to define $\boldsymbol{f}(\boldsymbol{x}, y)$ to contain conjunctions or other nonlinear combinations of the base features in $\boldsymbol{x}$. For example, a bigram feature such as ⟨*coffee house*⟩ will not fire unless both base features ⟨*coffee*⟩ and ⟨*house*⟩ also fire. More generally, we can define non-linear transformations such as the element-wise product $\boldsymbol{x} \circ \boldsymbol{x}$ and the cross-product $\boldsymbol{x} \otimes \boldsymbol{x}$.
- **Kernel** -based learning is based on similarity between instances; it can be seen as a generalization of k**-nearest-neighbors** , which classifies instances by considering the label of the k most similar instances in the training set (Hastie et al., 2009). The resulting decision boundary will be non-linear in general. Kernel functions can be designed to compute the similarity between structured objects, such as strings, bags-of-words, sequences, trees, and general graphs. Such methods will be discussed briefly in chapter 18.
- Boosting (Freund et al., 1999) and decision tree algorithms (Schmid, 1994) learn non-linear conjunctions of features. These methods sometimes do well on NLP tasks, but are used less frequently in contemporary research, especially as the field increasingly emphasizes big data and simple classifiers.
- More recent work has shown how **deep learning** can perform non-linear classification, by passing the inputs through a series of non-linear transformations. These methods will be reviewed in chapter 21; surveys are offered by Goldberg (2015) and Cho (2015).

[12] Assuming your feature matrix is full-rank.

	Naive Bayes	Logistic Regression	Perceptron	PA
Objective	Joint likelihood	Conditional likelihood	0/1 loss	Hinge loss
	$\max \sum_i \log \mathrm{p}(\boldsymbol{x}^{(i)}, y^{(i)})$	$\max \sum_i \log \mathrm{p}(y^{(i)} \mid \boldsymbol{x}^{(i)})$	$\min \sum_i \delta(y^{(i)}, \hat{y})$	$\sum_i [1 - \gamma(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})]_+$
estimation	$\theta_{ij} = \frac{c(x_i, y=j)+\alpha}{c(y=j)+V\alpha}$	$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \sum_i \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - E[\boldsymbol{f}(\boldsymbol{x}^{(i)}, y)]$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y})$	$\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} + \tau_t(\boldsymbol{f}(\boldsymbol{x}^{(i)}, y^{(i)}) - \boldsymbol{f}(\boldsymbol{x}^{(i)}, \hat{y}))$
tuning	smoothing α	regularizer $\lambda \|\boldsymbol{\theta}\|_2^2$	weight averaging	slack penalty C
complexity	$\mathcal{O}(NV)$	$\mathcal{O}(NVT)$	$\mathcal{O}(NVT)$	$\mathcal{O}(NVT)$
easy?	very	not really	yes	yes
probabilities?	yes	yes	no	no
features?	no	yes	yes	yes

Table 2.1: Comparison of classifiers. N = number of examples, V = number of features, T = number of instances.

(leave this page blank or the next page gets messed up)

Chapter 3

Linguistic applications of classification

Having learned some techniques for classification, let's now see how they can be applied to typical problems in natural language technology.

3.1 Sentiment and opinion analysis

A popular NLP technology is automatically determining the "sentiment" or "opinion polarity" of documents such as product reviews and social media posts. For example, marketers are interested to know how people respond to advertisements, services, and products (Hu and Liu, 2004); social scientists are interested in how emotions are affected by phenomena such as the weather (Hannak et al., 2012), and how both opinions and emotions spread over social networks (Coviello et al., 2014; Miller et al., 2011). In the field of **digital humanities**, literary scholars track plot structures through the flow of sentiment across a novel (Jockers, 2015). A comprehensive analysis of this broad literature is beyond the scope of this chapter, but see survey manuscripts by Pang and Lee (2008) and Liu (2015).

Sentiment analysis can be framed as a fairly direct application of document classification, assuming reliable labels can be obtained. In the simplest case, sentiment analysis can be treated as a two or three-class problem, with sentiments of POSITIVE, NEGATIVE, and possibly NEUTRAL. Such annotations could be annotated by hand, or obtained automatically through a variety of means:

- Tweets containing happy emoticons can be marked as positive, sad emoticons as negative (Read, 2005; Pak and Paroubek, 2010).
- Reviews with four or more stars can be marked as positive, two or fewer stars as negative (Pang et al., 2002).

- Statements from politicians who are voting **for** a given bill are marked as positive (towards that bill); statements from politicians voting against the bill are marked as negative (Thomas et al., 2006).

After obtaining the annotations, several design decisions may be taken in construction of the feature vector $\boldsymbol{f}(\boldsymbol{x}, y)$:

Preprocessing One question is whether the vocabulary should be case sensitive: do we distinguish *great*, *Great*, and GREAT? What about *cooooooool*? In social media text, this sort of **expressive lengthening** can cause the vocabulary size to explode (Brody and Diakopoulos, 2011); we might want to somehow **normalize** the text (Sproat et al., 2001) to collapse the vocabulary again.

A related issue is that suffixes may be irrelevant to the sentiment orientation of each word: for example, *love*, *loved*, and *loving* are all positive, so perhaps we should eliminate the suffix and group them together. The removal of these suffixes is called **stemming** when it is done at the character level (leaving roots like *lov-*), and is called **lemmatization** when the goal is to identify the underlying base word (in this case, *love*). Both of these methods will be discussed in detail in chapter 9 and chapter 8.

Still another preprocessing decision involves **tokenization**: breaking the text into tokens. This is more complicated than simply looking for whitespace, since we may want to tokenize items such as *well-bred* into $\langle$*well*, *bred*$\rangle$, *isn't* into $\langle$*is*, *n't*$\rangle$; at the same time, we would like to keep *U.S.* as a single token. This too will be discussed in chapter 8.

Vocabulary In some cases, it is preferable not to include all words in the vocabulary. Words such as *the*, *to*, and *and* seem intuitively to play little role in expressing sentiment or opinion, yet they are very frequent; removing these **stopwords** may therefore improve the classifier. This is typically done by creating a list and simply matching all items on the list. More aggressively, we might assume that sentiment is typically carried by **adjectives** and **adverbs** (see Chapter 7), and therefore we could focus on these words (Hatzivassiloglou and McKeown, 1997; Turney, 2002). However, Pang et al. (2002) find that in their case, eliminating non-adjectives causes the performance of the classifier to decrease.

Count or binary? Finally, we may consider whether we want our feature vector to include the **count** of each word, or its mere **presence**. This gets at a subtle limitation of linear classification: two *failure*s may be worse than one, but is it really twice as bad? A more flexible classifier could assign diminishing weight to each additional instance, but this is hard to do in the linear classification framework, and its hard to see how much the weight should diminish. Pang et al. (2002) take a simpler approach, using binary presence/absence indicators in the feature vector:

$f_i(\boldsymbol{x}, y) \in \{0, 1\}, \forall i$. They find that classifiers trained on these binary feature vectors outperform classifiers trained on count-based features.

A more challenging version of opinion analysis is to determine not just the class of a review, but its rating on a numerical scale (Pang and Lee, 2005). If the scale is continuous, we might take a regression approach, identifying a set of weights $\boldsymbol{\theta}$ so as to minimize the squared error of a predictor $\hat{y} = \boldsymbol{\theta}^\top \boldsymbol{x} + b$, where b is an offset. We can remove the offset by adding a feature to $\boldsymbol{x}$ whose value is always 1; the corresponding weight in $\boldsymbol{\theta}$ is then equivalent to b. Least squares regularization has a closed form solution,

$$\boldsymbol{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{y}, \tag{3.1}$$

where $\boldsymbol{y}$ is a column vector of size N, containing all ratings in the training data, and $\mathbf{X}$ is an $N \times D$ matrix containing all D features for all N instances. If we place an L2 regularizer on $\boldsymbol{\theta}$, with penalty $\lambda ||\boldsymbol{\theta}||_2^2$, the resulting problem is called **ridge regression**. It too has a closed form solution,

$$\boldsymbol{\theta} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbb{I})^{-1} \mathbf{X}^\top \boldsymbol{y}. \tag{3.2}$$

If the rating scale is discrete, $y \in \{1, 2, \ldots, K\}$, we can take a **ranking** approach (Crammer and Singer, 2001), in which scores $\boldsymbol{\theta}^\top \boldsymbol{x}$ are discretized into ranks, by also learning a set of boundaries, $b_0 = -\infty \leq b_1 \leq \ldots \leq b_K$. The learning algorithm consists in making perceptron-like updates to both $\boldsymbol{\theta}$ and $\boldsymbol{b}$. This approach is ideal for settings like predicting a 1-10 rating or a grade (A - F); instead of learning one vector $\boldsymbol{\theta}$ for every rank, we can learn a single $\boldsymbol{\theta}$, and then just partition the output space.

[todo: Other topics to cover:]

- subjectivity
- sentence-level versus document-level sentiment
- negation and the role of syntax
- targeted sentiment
- Stance classification

3.2 Word sense disambiguation

Consider the the following headlines:

(3.1) *Iraqi head seeks arms*

(3.2) *Prostitutes appeal to Pope*

(3.3) *Drunk gets nine years in violin case*[1]

They are ambiguous because they contain words that have multiple meanings, or **senses**. Word Sense Disambiguation (WSD) is the problem of identifying the intended sense of each word token in a document. WSD is part of a larger field of research called **lexical semantics**, which is concerned with meanings of the words.

Problem definition

Part-of-speech ambiguity (e.g., noun versus verb, as in *she is **heading** out of town*) is usually considered to be a different problem from WSD. Here we are focusing on ambiguity between senses that are all the same part-of-speech, and in part-of-speech tagging evaluations, it is often assumed that the correct part-of-speech has already been identified. [todo: why?] From a linguistic perspective, senses are not really properties of words, but of **lemmas**, which are groups of inflected forms, e.g. $(arm/N, arms/N)$, $(arm/V, arms/V, armed/V, arming/V)$, where arm/N indicates the word *arm* tagged as a noun (V is for verb). So the WSD problem can be defined as identifying the correct sense for each word token from an inventory associated for the word's lemma.

How many word senses?

Words (lemmas) may have *many* more than two senses. For example, the word *serve* would seem to have at least the following senses:

- [FUNCTION]: *The tree stump served as a table*
- [ENABLE]: *His evasive replies only served to heighten suspicion*
- [DISH]: *We serve only the rawest fish here*
- [ENLIST]: *She served her country in the marines*
- [JAIL]: *He served six years in Alcatraz*
- [TENNIS]: *Nobody can return his double-reverse spin serve*
- [LEGAL]: *They were served with subpoenas*[2]

How do we know that these senses are really different? Linguists often design tests for this purpose, and one such test is to construct a **zeugma**, which combines antagonistic senses in an uncomfortable way:

(3.4) *Which flight serves breakfast?*

[1] These examples, and many more, can be found at `http://www.ling.upenn.edu/~beatrice/humor/headlines.html`

[2] Examples from Dan Klein's lecture notes, `http://www.cs.berkeley.edu/~klein/cs294-7/SP07%20cs294%20lecture%205%20--%20maximum%20entropy%20(6pp).pdf`

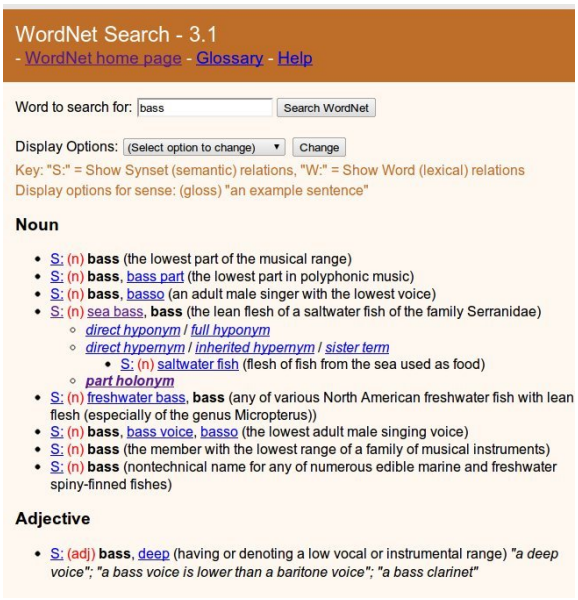

Figure 3.1: Example wordnet entry, from `http://wordnet.princeton.edu`

(3.5) *Which flights serve Tuscon?*

(3.6) **Which flights serve breakfast and Tuscon?*[3]

The asterisk is a linguistic notation for utterances which would not be judged to be grammatical by fluent speakers of a language. To the extent that you think that (3.6) is ungrammatical, you should agree that (3.4) and (3.5) refer to distinct senses of the lemma *serve*.

The WSD task: Output What should the output of WSD be? What are the possible senses for each word? We could just look in the dictionary. But rather than using a traditional dictionary, WSD research is dominated by a computational resource called WORDNET (`http://wordnet.princeton.edu`). WordNet is organized in terms of lemmas rather than words. An example of a wordnet entry is shown in Figure 3.1

WordNet consists of roughly 100,000 **synsets**, groups of words or phrases with an identical meaning. (e.g., $\{\text{CHUMP}^1, \text{FOOL}^2, \text{SUCKER}^1, \text{MARK}^9\}$). A lemma is **polysemous** if it participates in multiple synsets. Besides **synonymy**, WordNet also describes many other lexical relationships, including:

antonymy x means the opposite of y, e.g. FRIEND-ENEMY;

[3] I believe this example is from Jurafsky and Martin (2009) [todo: but check].

hyponymy x is a special case of y, e.g. RED-COLOR; the inverse relationship is **hypernymy**;

meronymy x is a part of y, e.g., WHEEL-BICYCLE; the inverse relationship is **holonymy**.

WordNet has played a big role in helping WSD move from toy systems to to large-scale quantitative evaluations. However, some have argued that WordNet's sense granularity is too fine (Ide and Wilks, 2006); more fundamentally, the premise that word senses can be differentiated in a task-neutral way has been criticized as linguistically naïve (Kilgarriff, 1997). One way of testing this question is to ask whether people tend to agree on the appropriate sense for example sentences: according to Mihalcea et al. (2004), humans agree on roughly 70% of examples using WordNet senses; far better than chance, but perhaps less than we might like.

A range of tasks have been proposed for WSD:

- **Synthetic** data: different words are conflated (*banana-phone*), the system must identify the original word.
- **Lexical sample**: disambiguate a few target words (e.g., *plant* etc). This is what was used in the first large-scale WSD evaluation, SENSEVAL-1 (1998).[todo: citation]
- **All-words** WSD: a sense must be identified for every token.
  - A **semantic concordance** is a corpus in which each open-class word (nouns, verbs, adjectives, and adverbs) is tagged with its word sense from the target dictionary or thesaurus.
  - SEMCOR is a semantic concordance built from 234K tokens of the Brown corpus.
    As of Sunday^1_n *night*^1_n *there was*^4_v *no word*^2_n ...

WSD as Classification

So, how can we tell living *plants* from manufacturing *plants*? The key information often lies in the **context**:

(3.7) *Town officials are hoping to attract new manufacturing plants through weakened environmental regulations.*

(3.8) *The endangered plant plays an important role in the local ecosystem.*

Bag-of-words models are a very typical approach. For example,

$$f(y, \textit{bank}, \textit{I went to the bank to deposit my paycheck}) = \{\langle \textit{went}, y\rangle : 1, \langle \textit{deposit}, y\rangle : 1, \langle \textit{paycheck}, y\rangle : 1\}$$

Some examples:[4]

- *bank*[FINANCIAL]:

  > a an and are ATM Bonnie card charges check Clyde criminals deposit famous for get I much My new overdraft really robbers the they think to too two went were

- *bank*[RIVER]:

  > a an and big campus cant catfish East got grandfather great has his I in is Minnesota Mississippi muddy My of on planted pole pretty right River The the there University walk Wets

An extension of bag-of-words models is to encode the position of each context word, e.g.,

$$f(y, \textit{bank}, \textit{I went to the bank to deposit my paycheck}) = \\ \{\langle i-3, \textit{went}, y\rangle : 1, \langle i+2, \textit{deposit}, y\rangle : 1, \langle i+4, \textit{paycheck}, y\rangle : 1\}$$

Jurafsky and Martin (2009) call these **collocation features**. Other approaches include more information about the sentence structure, such as the part-of-speech tag for each word, and the words with which it is syntactically linked in the sentence (see chapter 12).

After deciding on the features, we can train a classifier to predict the right sense of each word — assuming enough labeled examples can be accumulated. This is difficult, because each polysemous lemma requires its own training set: having a good classifier for *bank* is of no help at all towards disambiguating *plant*. For this reason, **unsupervised** and **semisupervised** methods are particularly popular for WSD (Yarowsky, 1995). We will talk about related methods in chapter 4 and chapter 20. Unsupervised methods typically lean heavily on the heuristic "one sense per discourse", meaning roughly that a lemma will have a consistent sense throughout any given document. Based on this heuristic, we can propagate information from high-confidence instances to lower-confidence instances in the same document. For a survey on word sense disambiguation, see Navigli (2009).

3.3 Other applications

- Author identification
- Author demographics, maybe
- Language classification

[4] todo: reconcile with examples above

3.4 Evaluating text classification

In any text classification setting, it is critical to reserve a held-out test set, and use this data for only one purpose: to evaluate the overall accuracy of a single classifier. Using this data more than once would cause your estimated accuracy to be overly optimistic. Since it is typically necessary to set hyperparameters or perform feature selection, you may need to construct various "tuning" or "development" sets, but these should not intersect with the test data. For more details, see section 1.2.

There are a number of ways to evaluate classifier performance. The simplest is **accuracy** : the number of correct predictions, divided by the total number of instances. Why isn't this always the right choice? Suppose we were building a classifier to detect whether an essay receives a passing grade. Due perhaps to grade inflation, 95% of all essays receive a passing grade. This means that a classifier that always says "pass" will get 95% accuracy. But this classifier isn't telling us anything useful at all.

Precision, recall, and F-measure

Another way to evaluate this classifier is in terms of its **precision** and **recall** . For each label $y \in \mathcal{Y}$, we define a **positive** instance as one that the classifier labels as $Y_i = y$, and a **negative** instance as one that the classifier labels as $Y_i \neq y$. We can then define four quantities:

True positive positive and correct, TP

False positive positive but incorrect, FP

True negative negative and correct, TN

False negative negative and incorrect, FN.

From these quantities, we can then define the **recall** and **precision**:

$$r = \frac{TP}{TP + FN} \tag{3.3}$$

$$p = \frac{TP}{TP + FP} \tag{3.4}$$

The recall is the proportion of positive labels among those that **should** have been labeled as positive (for some label y). The precision is the proportion of positive labels among those that **were** labeled as positive. Our "always pass" classifier above would have 100% recall for the positive label, but 95% precision. It would have 0% recall for the negative label, and undefined precision.

The **F-measure** is the harmonic mean of recall and precision,

$$F = \frac{2 \times r \times p}{r + p}. \tag{3.5}$$

F-measure is a classic measure of classifier performance for binary classification problems with unbalanced class distribution. Sometimes it is called F_1, as there are generalizations of F-measure in which the precision is multiplied by some constant β^2.

Macro-F_1 is the average F-measure across several classes. In a multi-class problem with unbalanced class distributions, the macro-F_1 is a balanced measure of how well the classifier recognizes each class. In **micro-F_1**, we compute true positives, false positives, and false negatives for each class, and then add them up before computing a single F-measure. This metric is balanced across instances rather than classes, so will weight each class in proportion to how frequently it appears.

Chapter 4

Learning without supervision

So far we've assumed the following setup:

- A **training set** where you get observations $\boldsymbol{x}_i$ and labels y_i
- A **test set** where you only get observations $\boldsymbol{x}_i$

What if you never get labels y_i? For example, suppose you are trying to do word sense disambiguation. You get a bunch of text, and you suspect that there are at least two different meanings for the word *concern*. But you don't have any labels for specific instances in which this word is used. What can you?

As described in chapter 3, in supervised word sense disambiguation, we often build feature vectors from the words that appear in the context of the word that we are trying to disambiguate. For example, for the word *concern*, the immediate context might typically include words from one of the following two groups:

1. *services, produces, banking, pharmaceutical, energy, electronics*

2. *about, said, that, over, in, with, had*

Now suppose we were to scatterplot each instance of *concern* on a graph, so that the x-axis is the density of words in group 1, and the y-axis is the density of words in group 2. In such a graph, shown in Figure 4.1, two or more blobs might emerge. These blobs would correspond to the different sense of *concern*.

But in reality, we don't know the word groupings in advance.[1] We have to try to apply the same idea in a very high dimensional space, where every word gets its own dimension — and most dimensions are irrelevant!

[1] One approach, which we do not consider here, would be to get them from some existing resource, such as the dictionary definition (Lesk, 1986).

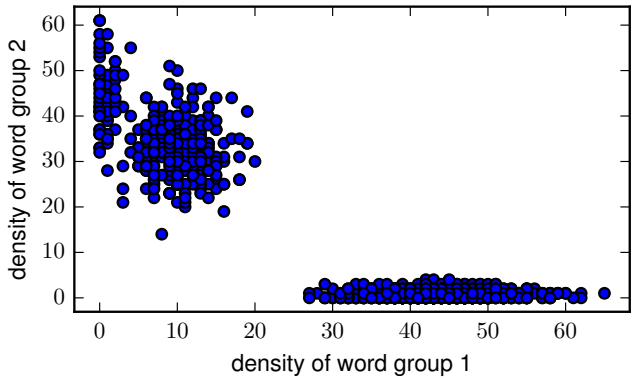

Figure 4.1: Counts of words from two different context groups

Now here's a related scenario, from a different problem. Suppose you download thousands of news articles, and make a scatterplot, where each point corresponds to a document: the x-axis is the frequency of the word *hurricane*, and the y-axis is the frequency of the word *election*. Again, three clumps might emerge: one for documents that are largely about the hurricane, another for documents largely about the election, and a third clump for documents about neither topic.

These examples are intended to show that we can find structure in data, even without labels — just look for clumps in the scatterplot of features. But again, in reality we cannot make scatterplots of just two words; we may have to consider hundreds or thousands of words. It would be impossible to visualize such a high-dimensional scatterplot, so we will need to design algorithmic approaches to finding these groups.

4.1 K-means clustering

You might know about classic clustering algorithms like **K-means**. These algorithms maintain a cluster assignment for each instance, and a central location for each cluster. They them repeatedly update the cluster assignments and the locations, until convergence. Pseudocode for K-means is shown in Algorithm 4.

K-means can used to find coherent clusters of documents in high-dimensional data. When we assign each point to its nearest center, we are choosing which cluster it is in; when we re-estimate the location of the centers, we are determining the defining characteristic of each cluster. K-means is a classic algorithmic that has been used and modified in thousands of papers (Jain, 2010); for an application of K-means to word sense induction, see Pantel and Lin (2002).

Of the many variants of K-means, one that is particularly relevant for our purposes is called **soft K-means**. The key difference is that instead of directly assigning each point $\boldsymbol{x}_i$

Algorithm 4 K-means clustering algorithm

procedure K-MEANS($\boldsymbol{x}_{1:N}$)
  Initialize cluster centers $\mu_k \leftarrow \text{Random}()$
  repeat
    for all i **do**
      Assign each point to the nearest cluster: $z_i \leftarrow \min_k \text{Distance}(\boldsymbol{x}_i, \mu_k)$
    for all k **do**
      Recompute each cluster center from the points in the cluster: $\mu_k \leftarrow \frac{1}{\sum_i \delta(z_i=k)} \sum_i \delta(z_i = k)\boldsymbol{x}_i$
  until converged

to a specific cluster z_i, soft K-means assigns each point a **distribution** over clusters $q_i(z_i)$, so that $\sum_k q_i(k) = 1$, and $\forall_k 0 \leq q_i(k) \leq 1$. The centroid of each cluster is then computed from a **weighted average** of the points in the cluster, where the weights are taken from the q distribution.

We will now explore a more principled, statistical version of soft K-means, called **expectation maximization** (EM) clustering. By understanding the statistical principles underlying the algorithm, we can extend it in a number of ways.

4.2 The Expectation Maximization (EM) Algorithm

Let's go back to the Naïve Bayes model:

$$\log \text{p}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{\phi}, \mu) = \sum_i \log \text{p}(\boldsymbol{x}_i \mid y_i; \boldsymbol{\phi}) \text{p}(y_i; \mu) \tag{4.1}$$

For example, $\boldsymbol{x}$ can describe the documents that we see today, and $\boldsymbol{y}$ can correspond to their labels. But suppose we never observe y_i? Can we still do anything with this model?

Since we don't know $\boldsymbol{y}$, let's marginalize it:

$$\log \text{p}(\boldsymbol{x}) = \sum_i^N \log \text{p}(\boldsymbol{x}_i) \tag{4.2}$$

$$= \sum_i \log \sum_{y_i} \text{p}(\boldsymbol{x}_i \mid y_i; \boldsymbol{\phi}) \text{p}(y_i; \mu) \tag{4.3}$$

$$\tag{4.4}$$

We will estimate the parameters $\boldsymbol{\phi}$ and μ by maximizing the log-likelihood of $\boldsymbol{x}_{1:N}$, which is our (unlabeled) observed data. Why is this a good thing to maximize? If we

don't have labels, discriminative learning is impossible (there's nothing to discriminate), so maximum likelihood is all we have.

Unfortunately, maximizing $\log P(\boldsymbol{x})$ directly is intractable. So to estimate this model, we must employ approximation. We do this by introducing an **auxiliary variable** $\boldsymbol{q}_i$, for each y_i. We want $\boldsymbol{q}_i$ to be a **distribution**, so we have the usual constraints: $\sum_y q_i(y) = 1$ and $\forall y, q_i(y) \geq 0$. In other words, q_i defines a probability distribution over $\mathcal{Y}$, for each instance i.

Now since $\frac{q_i(y)}{q_i(y)} = 1$, we can multiply the right side by this ratio and preserve the equality,

$$\log \mathrm{p}(\boldsymbol{x}) = \sum_i \log \sum_{y_i} \mathrm{p}(\boldsymbol{x}_i \mid y_i; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu) \frac{q_i(y)}{q_i(y)} \tag{4.5}$$

$$= \sum_i \log E_q \left[\frac{\mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) \mathrm{p}(y; \mu)}{q_i(y)} \right], \tag{4.6}$$

by the definition of expectation, $E_q[f(x)] = \sum_x q(x) f(x)$. Note that $E_q[\cdot]$ just means the expectation under the distribution q.

Now we apply **Jensen's inequality**, which says that because $\log$ is a concave function, we can push it inside the expectation, and obtain a lower bound.

$$\log \mathrm{p}(\boldsymbol{x}) \geq \sum_i E_q \left[\log \frac{\mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) \mathrm{p}(y_i; \mu)}{q_i(y)} \right] \tag{4.7}$$

$$\mathcal{J} = \sum_i E_q [\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi})] + E_q [\log \mathrm{p}(y; \mu)] - E_q [\log q_i(y)] \tag{4.8}$$

By maximizing $\mathcal{J}$, we are maximizing a lower bound on the joint log-likelihood $\log \mathrm{p}(\boldsymbol{x})$. Now, $\mathcal{J}$ is a function of two sets of arguments:

- the distributions q_i for each i
- the parameters μ and $\boldsymbol{\phi}$

We'll optimize with respect to each of these in turn, holding the other one fixed.

Step 1: the E-step

First, we expand the expectation in the lower bound as:

$$\mathcal{J} = \sum_i E_q [\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi})] + E_q [\log \mathrm{p}(y; \mu)] - E_q [\log q_i(y)] \tag{4.9}$$

$$= \sum_i \sum_y q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y) \right) \tag{4.10}$$

As in Naïve Bayes, we have a "sum-to-one" constraint: in this case, $\sum_y q_i(y) = 1$. Once again, we incorporate this constraint into a Lagrangian:

$$\mathcal{J}_q = \sum_i^N \sum_{y \in \mathcal{Y}} q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right) + \lambda_i (1 - \sum_y q_i(y)) \tag{4.11}$$

We then optimize by taking the derivative and setting it equal to zero:

$$\frac{\partial \mathcal{J}_q}{\partial q_i(y)} = \log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \boldsymbol{\theta}) - \log q_i(y) - 1 - \lambda_i \tag{4.12}$$

$$\log q_i(y) = \log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - 1 - \lambda_i \tag{4.13}$$

$$q_i(y) \propto \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) \mathrm{p}(y; \mu) = \mathrm{p}(\boldsymbol{x}_i, y; \boldsymbol{\phi}, \mu) \tag{4.14}$$

Since q_i is defined over the labels $\mathcal{Y}$, we normalize it as,

$$q_i(y) = \frac{\mathrm{p}(\boldsymbol{x}_i, y; \boldsymbol{\phi}, \mu)}{\sum_{y' \in \mathcal{Y}} \mathrm{p}(\boldsymbol{x}_i, y'; \boldsymbol{\phi}, \mu)} = \mathrm{p}(y \mid \boldsymbol{x}_i; \boldsymbol{\phi}, \mu) \tag{4.15}$$

After normalizing, each $q_i(y)$ — which is the soft distribution over clusters for data $\boldsymbol{x}_i$ — is set to the posterior probability $\mathrm{p}(y \mid \boldsymbol{x}_i)$ under the current parameters $\mu, \boldsymbol{\phi}$. This is called the E-step, or "expectation step," because it is derived from updating the bound on the expected likelihood under $q(\boldsymbol{y})$. Note that although we introduced the Lagrange multipliers λ_i as additional parameters, we were able to drop these parameters because we solved for $q_i(y)$ to a constant of proportionality.

Step 2: the M-step

Next, we hold $q(\boldsymbol{y})$ fixed and maximize the bound with respect to the parameters, $\boldsymbol{\phi}$ and μ. Lets focus on $\boldsymbol{\phi}$, which parametrizes the likelihood, $\mathrm{p}(\boldsymbol{x} \mid y; \boldsymbol{\phi})$. Again, we have a constraint that $\sum_j^V \phi_{y,j} = 1$, so we start by forming a Lagrangian,

$$\mathcal{J}_\phi = \sum_i^N \sum_{y \in \mathcal{Y}} q_i(y) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid y; \boldsymbol{\phi}) + \log \mathrm{p}(y; \mu) - \log q_i(y)\right) + \sum_{y \in \mathcal{Y}} \lambda_y (1 - \sum_j^V \phi_{y,j}). \tag{4.16}$$

Again, we solve by setting the derivative equal to zero:

$$\frac{\partial \mathcal{J}_\phi}{\partial \phi_{y,j}} = \sum_i^N q_i(y) \frac{x_{i,j}}{\phi_{y,j}} - \lambda_y \tag{4.17}$$

$$\lambda_h \phi_{y,j} = \sum_i^N q_i(y) x_{i,j} \tag{4.18}$$

$$\phi_{y,j} \propto \sum_i^N q_i(y) x_{i,j}. \tag{4.19}$$

Now because $\sum_j^V \phi_{y,j} = 1$, we can normalize as follows,

$$\phi_{y,j} = \frac{\sum_i^N q_i(y) x_{i,j}}{\sum_{j'<V} \sum_i^N q_i(y) x_{i,j'}} \tag{4.20}$$

$$= \frac{E_q\left[\text{count}(y, j)\right]}{E_q\left[\text{count}(y)\right]}, \tag{4.21}$$

where $j \in \{1, 2, \ldots, V\}$ indexes base features, such as words.

So $\boldsymbol{\phi}_y$ is now equal to the relative frequency estimate of the **expected counts** under the distribution $q(y)$.

- As in supervised Naïve Bayes, we can apply smoothing to add α to all these counts.
- The update for μ is identical: $\mu_y \propto \sum_i q_i(y)$, the expected proportion of cluster $Y = y$. If needed, we can add smoothing here too.
- So, everything in the M-step is just like Naïve Bayes, except that we use expected counts rather than observed counts.

This is the M-step for a model in which the likelihood $P(\boldsymbol{x} \mid \boldsymbol{y})$ is multinomial. For other likelihoods, there my be no closed-form solution for the parameters in the M-step. We may therefore run gradient-based optimization at each M-step, or we may simply take a single step along the gradient step and then return to the E-step (Berg-Kirkpatrick et al., 2010).

Coordinate ascent

Algorithms that alternate between updating various subsets of the parameters are called "coordinate ascent" algorithms.

The objective function $\mathcal{J}$ is **biconvex**, meaning that it is separately convex in $q(\boldsymbol{y})$ and $\langle \mu, \boldsymbol{\phi} \rangle$, but it is not jointly convex in all terms. In the coordinate ascent algorithm that we have defined, each step is guaranteed not to decrease $\mathcal{J}$. This is sometimes called "hill climbing", because you never go down. Specifically, EM is guaranteed to converge to a **local optima** — a point which is as good or better than any of its immediate neighbors. But there may be many such points, and the overall procedure is **not** guaranteed to find a global maximum. Figure 4.2 shows the objective function for EM with ten different random initializations: while the objective function increases monotonically in each run, it converges to several different values.

The fact that there is no guarantee of global optimality means that initialization is important: where you start can determine where you finish. This is not true in the supervised learning algorithms that we have considered, such as logistic regression — although deep learning algorithms do suffer from this problem. But for logistic regression, and for many other supervised learning algorithms, we don't need to worry about initialization,

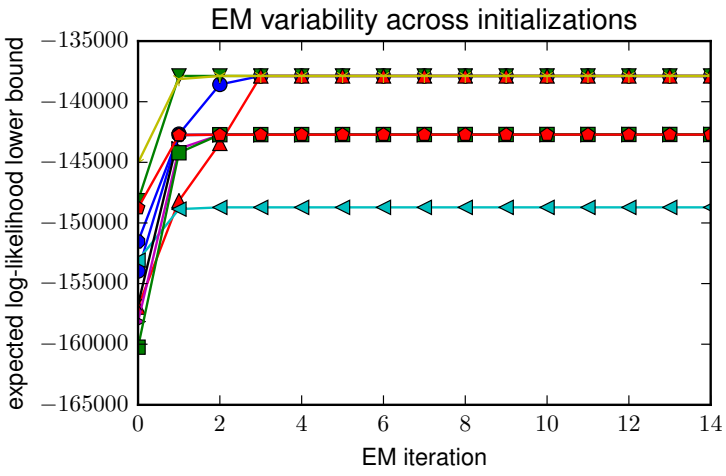

Figure 4.2: Sensitivity of expectation maximization to initialization

because it won't affect our ultimate solution: we are guaranteed to reach the global minimum. Recent work on **spectral learning** has sought to obtain similar guarantees for "latent variable" models, such as the case we are considering now, where $\boldsymbol{x}$ is observed and y is latent. This work is briefly touched on in section 4.4.

Variants In **hard EM** , each q_i distribution assigns probability of 1 to a single $\hat{y}_i$, and probability of 0 to all others (Neal and Hinton, 1998). This is similar in spirit to K-means clustering. In problems where the space $\mathcal{Y}$ is large, it may be easier to find the maximum likelihood value $\hat{y}$ than it is to compute the entire distribution $q_i(y)$. Spitkovsky et al. (2010) show that hard EM can outperform standard EM in some cases.

Another variant of the coordinate ascent procedure combines EM with stochastic gradient descent (SGD). In this case, we can do a local E-step at each instance i, and then immediately make an gradient update to the parameters $\langle \mu, \boldsymbol{\phi} \rangle$. This is particularly relevant in cases where there is no closed form solution for the parameters, so that gradient ascent will be necessary in any case. This algorithm is called "incremental EM" by Neal and Hinton (1998), and online EM by Sato and Ishii (2000) and Cappé and Moulines (2009). Liang and Klein (2009) apply a range of different online EM variants to NLP problems, obtaining better results than standard EM in many cases.

How many clusters?

All along, we have assumed that the number of clusters $K = \#|\mathcal{Y}|$ is given. In some cases, this assumption is valid. For example, the dictionary or WordNet might tell us the number of senses for a word. In other cases, the number of clusters should be a tunable

parameter: some readers may want a coarse-grained clustering of news stories into three or four clusters, while others may want a fine-grained clusterings into twenty or more. But in many cases, we will have choose K ourselves, with little outside guidance.

One solution is to choose the number of clusters to maximize some computable quantity of the clustering. First, note that the likelihood of the training data will always increase with K. For example, if a good solution is available for $K = 2$, then we can always obtain that same solution at $K > 2$; usually we can find an even better solution by fitting the data more closely. The Akaike Information Crition (AIC; Akaike, 1974) solves this problem by minimizing a linear combination of the log-likelihood and the number of model parameters, AIC $= 2m - 2\mathcal{L}$, where m is the number of parameters and $\mathcal{L}$ is the log-likelihood. Since the number of parameters increases with the number of clusters K, the AIC may prefer more parsimonious models, even if they do not fit the data quite as well.

Another choice is to maximize the **predictive likelihood** on heldout data $\boldsymbol{x}^{(h)}_{1:N_h}$. This data is not used to estimate the model parameters $\boldsymbol{\phi}$ and μ; we can compute the predictive likelihood on this data by keeping the parameters $\boldsymbol{\phi}$ and μ fixed, and running a single iteration of the E-step. In document clustering or **topic modeling** (Blei, 2012), a typical approach is to split each instance (document) in half. We use the first half to estimate $q_i(z_i)$, and then on the second half we compute the expected log-likelihood,

$$\ell_i = \sum_z q_i(z) \left(\log \mathrm{p}(\boldsymbol{x}_i \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z; \mu) \right). \tag{4.22}$$

On heldout data, this quantity will not necessarily increase with the number of clusters K, because for high enough K, we are likely to overfit the training data. Thus, choosing K to maximize the predictive likelihood on heldout data will limit the extent of overfitting. Note that in general we cannot analytically find the K that maximizes either AIC or the predictive likelihood, so we must resort to grid search: trying a range of possible values of K, and choosing the best one.

Finally, it is worth mentioning an alternative approach, called **Bayesian nonparametrics**, in which the number of clusters K is treated as another latent variable. This enables statistical inference over a set of models with a variable number of clusters; this is not possible with EM, but there are several alternative inference procedures that are suitable for this case (Murphy, 2012), including MCMC (section 4.4). Reisinger and Mooney (2010) provide a nice example of Bayesian nonparametrics in NLP, applying it to unsupervised word sense induction.

4.3 Applications of EM

EM is not really an "algorithm" like, say, quicksort. Rather, it is a framework for learning with missing data. The recipe for using EM on a problem of interest is:

- Introduce latent variables $\boldsymbol{z}$, such that it is easy to write the probability $P(\mathcal{D}, \boldsymbol{z})$, where $\mathcal{D}$ is your observed data; it should also be easy to estimate the associated parameters, given knowledge of $\boldsymbol{z}$.
- Derive the E-step updates for $q(\boldsymbol{z})$, which is typically factored as $q(\boldsymbol{z}) = \prod_i q_{z_i}(z_i)$, where i is an index over instances.
- The M-step updates typically correspond to the soft version of some supervised learning algorithm, like Naïve Bayes.

Some more applications of this basic setup are presented here.

Word sense clustering

In the "demos" folder, you can find a demonstration of expectation maximization for word sense clustering. I assume we know that there are two senses, and that the senses can be distinguished by the contextual information in the document. The basic framework is identical to the clustering model of EM as presented above.

Semi-supervised learning

Nigam et al. (2000) offer another application of EM: **semi-supervised learning**. They apply this idea to document classification in the classic "20 Newsgroup" dataset, in which each document is a post from one of twenty newsgroups from the early days of the internet.

In the setting considered by Nigam et al. (2000), we have labels for some of the instances, $\langle \boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)} \rangle$, but not for others, $\langle \boldsymbol{x}^{(u)} \rangle$. The question they pose is: can unlabeled data improve learning? If so, then we might be able to get good performance from a smaller number of labeled instances, simply by incorporating a large number of unlabeled instances. This idea is called **semi-supervised learning**, because we are learning from a combination of labeled and unlabeled data; the setting is described in much more detail in chapter 20.

As in Naïve Bayes, the learning objective is to maximize the joint likelihood,

$$\log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{x}^{(u)}, \boldsymbol{y}^{(\ell)}) = \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \log \mathrm{p}(\boldsymbol{x}^{(u)}) \tag{4.23}$$

We treat the labels of the unlabeled documents as missing data — in other words, as a latent variable. In the E-step we impute $q(y)$ for the unlabeled documents only. The M-step computes estimates of μ and $\boldsymbol{\phi}$ from the sum of the observed counts from $\langle \boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)} \rangle$ and the expected counts from $\langle \boldsymbol{x}^{(u)} \rangle$ and $q(\boldsymbol{y})$.

Nigam et al. (2000) further parametrize this approach by weighting the unlabeled documents by a scalar λ, which is a tuning parameter. The resulting criterion is:

$$\mathcal{L} = \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \lambda \log \mathrm{p}(\boldsymbol{x}^{(u)}) \tag{4.24}$$

$$\geq \log \mathrm{p}(\boldsymbol{x}^{(\ell)}, \boldsymbol{y}^{(\ell)}) + \lambda E_q[\log \mathrm{p}(\boldsymbol{x}^{(u)}, y)] \tag{4.25}$$

The scaling factor does not really have a probabilistic justification, but it can be important to getting good performance, especially when the amount of labeled data is small in comparison to the amount of unlabeled data. In that scenario, the risk is that the unlabeled data will dominate, causing the parameters to drift towards a "natural clustering" that may be a bad fit for the labeled data. Nigam et al. (2000) show that this approach can give substantial improvements in classification performance when the amount of labeled data is small.

Multi-component modeling

Now let us consider an alternative application of EM to supervised classification. One of the classes in 20 newsgroups is `comp.sys.mac.hardware`; suppose that within this newsgroup there are two kinds of posts: reviews of new hardware, and question-answer posts about hardware problems. The language in these **components** of the mac.hardware class might have little in common. So we might do better if we model these components separately. Nigam et al. (2000) show that EM can be applied to this setting as well.

Recall that Naïve Bayes is based on a generative process, which provides a stochastic explanation for the observed data. For multi-component modeling, we envision a slightly different generative process, incorporating both the observed label y_i and the latent component z_i:

- For each document i,
  - draw the label $y_i \sim \text{Categorical}(\mu)$
  - draw the component $z_i \mid y_i \sim \text{Categorical}(\beta_{y_i})$, where $z_i \in 1, 2, \ldots, K_z$.
  - draw the vector of counts $\boldsymbol{x}_i \mid z_i \sim \text{Multinomial}(\boldsymbol{\phi}_{z_i})$

Our labeled data includes $\langle \boldsymbol{x}_i, y_i \rangle$, but not z_i, so this is another case of missing data. Again, we sum over the missing data, applying Jensen's inequality to as to obtain a lower bound on the log-likelihood,

$$\log \mathrm{p}(\boldsymbol{x}_i, y_i) = \log \sum_{z}^{K_z} \mathrm{p}(\boldsymbol{x}_i, y_i, z) \tag{4.26}$$

$$\geq \log \mathrm{p}(y_i; \mu) + E_q \left[\log \mathrm{p}(\boldsymbol{x}_i \mid z; \boldsymbol{\phi}) + \log \mathrm{p}(z \mid y_i; \psi) - \log q_i(z)\right]. \tag{4.27}$$

We are now ready to apply expectation maximization. As usual, the distribution over the missing data — the component z_i — $q_i(z)$ is updated in the E-step. Then during the m-step, we compute:

$$\beta_{y,z} = \frac{E_q\left[\text{count}(y,z)\right]}{\sum_{z'}^{K_z} E_q\left[\text{count}(y,z')\right]} \tag{4.28}$$

$$\boldsymbol{\phi}_{z,j} = \frac{E_q\left[\text{count}(z,j)\right]}{\sum_{j'}^{V} E_q\left[\text{count}(z,j')\right]}. \tag{4.29}$$

Suppose we assume each class y is associated with K components, $\mathcal{Z}_y$. We can then add a constraint to the E-step so that $q_i(z) = 0$ if $z \notin \mathcal{Z}_y \wedge Y_i = y$.

4.4 Other approaches to learning with latent variables*

Expectation maximization is a very general way to think about learning with latent variables, but it has some limitations. One is the sensitivity to initialization, which means that we cannot simply run EM once and expect to get a good solution. Indeed, in practical applications of EM, quite a lot of attention may be devoted to finding a good initialization. A second issue is that EM tends to be easiest to apply in cases where the latent variables have a clear decomposition (in the cases we have considered, they decompose across the instances). For these reasons, it is worth briefly considering some alternatives to EM.

Sampling

Recall that in EM, we set $q(\boldsymbol{z}) = \prod_i q_i(z_i)$, factoring the q distribution into conditionally independent q_i distributions. In sampling-based algorithms, rather than maintaining a distribution over each latent variable, we draw random samples of the latent variables. If the sampling algorithm is designed correctly, this procedure will eventually converge to drawing samples from the true posterior, $\text{p}(\boldsymbol{z}_{1:N} \mid \boldsymbol{x}_{1:N})$. For example, in the case of clustering, we will draw samples from the distribution over clusterings of the data. If a single clustering is required, we can select the one with the highest joint likelihood, $\text{p}(\boldsymbol{z}_{1:N}, \boldsymbol{x}_{1:N})$.

This general family of algorithms is called **Markov Chain Monte Carlo** (MCMC): "Monte Carlo" because it is based on a series of random draws; "Markov Chain" because the sampling procedure must be designed such that each sample depends only on the previous sample, and not on the entire sampling history. Gibbs Sampling is a particularly simple and effective MCMC algorithm, in which we sample each latent variable from its posterior distribution,

$$z_i \mid \boldsymbol{x}, \boldsymbol{z}_{-i} \sim \text{p}(z_i \mid \boldsymbol{x}, \boldsymbol{z}_{-i}), \tag{4.30}$$

where $\boldsymbol{z}_{-i}$ indicates $\{\boldsymbol{z} \backslash z_i\}$, the set of all latent variables except for z_i.

What about the parameters, ϕ and μ? One possibility is to turn them into latent variables too, by adding them to the generative story. This requires specifying a prior distribution; the Dirichlet is a typical choice of prior for the parameters of a multinomial, since it has support over vectors of non-negative numbers that sum to one, which is exactly the set of permissible parameters for a multinomial. For example,

$$\boldsymbol{\phi}_y \sim \text{Dirichlet}(\alpha), \forall y \tag{4.31}$$

We can then sample $\boldsymbol{\phi}_y \mid \boldsymbol{x}, \boldsymbol{z} \sim \text{p}(\boldsymbol{\phi}_y \mid \boldsymbol{x}, \boldsymbol{z}, \alpha)$; this posterior distribution will also be Dirichlet, with parameters $\alpha + \sum_{i:y_i=y} \boldsymbol{x}_i$. Alternatively, we can analytically marginalize these parameters, as in **Collapsed Gibbs Sampling**; this is usually preferable if possible. Finally, we might maintain $\boldsymbol{\phi}$ and μ as parameters rather than latent variables. We can employ sampling in the E-step of the EM algorithm, obtaining a hybrid algorithm called Monte Carlo Expectation Maximization (MCEM; Wei and Tanner, 1990).

In principle, these algorithms will eventually converge to the true posterior distribution. However, there is no way to know how long this will take; there is not even any way to check on whether the algorithm has converged. In practice, convergence again depends on initialization, since it might take ages to recover from a poor initialization. Thus, while Gibbs Sampling and other MCMC algorithms provide a powerful and flexible array of techniques for statistical inference in latent variable models, they are not a panacea for the problems experienced by EM.

Murphy (2012) includes an excellent chapter on MCMC; for a more comprehensive treatment, see Robert and Casella (2013).

Spectral learning

A more recent approach to learning with latent variables is based on the **method of moments**. In these approaches, we avoid the problem of non-convex log-likelihood by using a different estimation criterion. Let us write $\overline{\boldsymbol{x}}_i$ for the normalized vector of word counts in document i, so that $\overline{\boldsymbol{x}}_i = \boldsymbol{x}_i / \sum_j x_{ij}$. Then we can form a matrix of word-word co-occurrence counts,

$$\mathbf{C} = \sum_i \boldsymbol{x}_i \boldsymbol{x}_i^\top. \tag{4.32}$$

We can also compute the expected value of this matrix under $\text{p}(\boldsymbol{x} \mid \boldsymbol{\phi}, \mu)$, as

$$E[\mathbf{C}] = \sum_i \sum_k P(Z_i = k \mid \mu) \boldsymbol{\phi}_k \boldsymbol{\phi}_k^\top \tag{4.33}$$

$$= \sum_k N \mu_k \boldsymbol{\phi}_k \boldsymbol{\phi}_k^\top \tag{4.34}$$

$$= \Phi \text{Diag}(N\mu) \Phi^\top, \tag{4.35}$$

where Φ is formed by horizontally concatenating $\boldsymbol{\phi}_1 \ldots \boldsymbol{\phi}_K$, and Diag$(N\mu)$ indicates a diagonal matrix with values $N\mu_k$ at position (k, k). Now, by setting $\mathbf{C}$ equal to its expectation, we obtain,

$$\mathbf{C} = \boldsymbol{\Phi}\text{Diag}(N\mu)\boldsymbol{\Phi}^\top, \tag{4.36}$$

which is very similar to the eigendecomposition $\mathbf{C} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top$. This suggests that simply by finding the eigenvectors and eigenvalues of $\mathbf{C}$, we could obtain the parameters $\boldsymbol{\phi}$ and μ, and this is what motivates the name **spectral learning**.

However, there is a key difference in the constraints on the solutions to the two problems. In eigendecomposition, we require orthonormality, so that $\mathbf{Q}\mathbf{Q}^\top = \mathbb{I}$. But in estimating the parameters of a mixture model, we require the columns of Φ represents probability vectors, $\forall k, j, \phi_{k,j} \geq 0, \sum_j \phi_{k,j} = 1$, and that the entries of μ correspond to the probabilities over components. Thus, spectral learning algorithms must include a procedure for converting the solution into vectors of probabilities. One approach is to replace eigendecomposition (or the related singular value decomposition) with non-negative matrix factorization (Xu et al., 2003), which guarantees that the solutions are non-negative (Arora et al., 2013).

After obtaining the parameters $\boldsymbol{\phi}$ and μ, we can obtain the distribution over clusters for each document by simply computing $\text{p}(z_i \mid \boldsymbol{x}_i; \boldsymbol{\phi}, \mu) \propto \text{p}(\boldsymbol{x}_i \mid z_i; \boldsymbol{\phi})\text{p}(z_i; \mu)$. The advantages of spectral learning are that it obtains (provably) good solutions without regard to initialization, and that it can be quite fast in practice. Anandkumar et al. (2014) describe how similar matrix and tensor factorizations can be applied to statistical estimation in many other forms of latent variable models.

Chapter 5

Language models

In probabilistic classification, we are interested in computing the probability of a label, conditioned on the text. Let us now consider something like the inverse problem: computing the probability of text itself. Specifically, we will consider models that assign probability to a sequence of word tokens,[1] $\text{p}(w_1, w_2, \ldots, w_M)$, with $w_m \in \mathcal{V}$. The set $\mathcal{V}$ is a discrete vocabulary,

$$\mathcal{V} = \{\textit{aardvark}, \textit{abacus}, \ldots, \textit{zither}\}. \tag{5.1}$$

Why would we want to compute the probaibility of a word sequence? In many applications, our goal is to produce word sequences as output:

- In **machine translation**, we convert from text in a source language to text in a target language.
- In **speech recognition**, we convert from audio signal to text.
- In **summarization**, we convert from long texts into short texts.
- In **dialogue systems**, we convert from the user's input (and perhaps an external knowledge base) into a text response.

In each of these cases, a key subcomponent is to compute the probability of the output text. By choosing high-probability output, we hope to generate texts that are more **fluent** . For example, suppose we want to translate a sentence from Spanish to English.

(5.1) *El cafe negro me gusta mucho.*

[1] The linguistic term "word" does not cover everything we might want to model, such as names, numbers, and emoticons. Instead, we prefer the term **token** , which refers to anything that can appear in a sequence of linguistic data. **Tokenizers** are programs for segmenting strings of characters or bytes into tokens. In English, tokenization is relatively straightforward, and can be performed using a regular expression. But in languages like Chinese, tokens are not usually separated by spaces, so tokenization can be considerably more challenging. For more on tokenization algorithms, see Manning et al. (2008), chapter 2.

The literal word-for-word translation (sometimes called a **gloss**) is,

(5.2) *The coffee black me pleases much.*

A good language model of English will tell us that the probability of this translation is low. Furthermore,

$$\text{p}(\textit{The coffee black me pleases much}) < \text{p}(\textit{I love dark coffee}). \tag{5.2}$$

How can we use this fact? Warren Weaver, one of the early leaders in machine translation, viewed it as a problem of breaking a secret code (Weaver, 1955):

> When I look at an article in Russian, I say: 'This is really written in English, but it has been coded in some strange symbols. I will now proceed to decode.'

This observation motivates a generative model (like Naïve Bayes):

- The English sentence $\boldsymbol{w}^{(e)}$ is generated from a **language model** , $\text{p}_e(\boldsymbol{w}^{(e)})$.
- The Spanish sentence $\boldsymbol{w}^{(s)}$ is then generated from a **translation model** $\text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)})$.

Given these two distributions, we can then perform translation by Bayes rule:

$$\begin{aligned}\text{p}_{e|s}(\boldsymbol{w}^{(e)} \mid \boldsymbol{w}^{(s)}) \propto & \text{p}_{e,s}(\boldsymbol{w}^{(e)}, \boldsymbol{w}^{(s)}) && (5.3)\\ = & \text{p}_{s|e}(\boldsymbol{w}^{(s)} \mid \boldsymbol{w}^{(e)}) \times \text{p}_e(\boldsymbol{w}^{(e)}). && (5.4)\end{aligned}$$

This is sometimes called the **noisy channel model** , because it envisions English text turning into Spanish by passing through a noisy channel, $\text{p}_{s|e}$. What is the advantage of modeling translation this way, as opposed to modeling $\text{p}_{e|s}$ directly? The crucial point is that the two distribtions $\text{p}_{s|e}$ (the translation model) and p_e (the language model) can be estimated from separate data. The translation model requires **bitext** — examples of correct translations. But the language model requires only text in English. Such monolingual data is much more widely available, which means that the fluency of the output translation can be improved simply by scraping more webpages. Furthermore, once estimated, the language model p_e can be reused in any application that involves generating English text, from summarization to speech recognition.

5.1 N-gram language models

How can we estimate the probability of a sequence of word tokens? The simplest idea would be to apply a **relative frequency estimator**. For example, consider the quote, attributed to Picasso, *Computers are useless, they can only give you answers.* We can estimate

the probability of this sentence as follows:

$$\text{p}(\textit{Computers are useless, they can only give you answers}) \tag{5.5}$$
$$= \frac{\text{count}(\textit{Computers are useless, they can only give you answers})}{\text{count}(\text{all sentences ever spoken})} \tag{5.6}$$

It is useful to think about this estimator in terms of bias and variance. In the theoretical limit of infinite data, this could work. But in practice, we are asking for accurate counts over an infinite number of events, since sequences of words can be arbitrarily long. Even if we set an aggressive upper bound of, say, $n = 20$ tokens in the sequence, the number of possible sequences is $|\mathcal{V}|^{20}$. A small vocabularly for English would have $|\mathcal{V}| = 10^4$, so we would have 10^{80} possible sequences. Clearly, this estimator is very data-hungry, and suffers from high variance: even grammatical sentences will have probability zero if they happen not to have occurred in the training data.[2] We therefore need to introduce bias to have a chance of making reliable estimates from finite training data. The language models that follow in this chapter introduce bias in various ways.

We begin with n-gram language models, which compute the probability of a sequence as the product of probabilities of subsequences. The probability of a sequence $\text{p}(\boldsymbol{w}) = \text{p}(w_1, w_2, \ldots, w_M)$ can be refactored using the chain rule:

$$\begin{aligned}\text{p}(\boldsymbol{w}) =& \text{p}(w_1, w_2, \ldots, w_M) \\ =& \text{p}(w_1) \times \text{p}(w_2 \mid w_1) \times \text{p}(w_3 \mid w_2, w_1) \times \ldots \times \text{p}(w_M \mid w_{M-1}, \ldots, w_1)\end{aligned}$$

Each element in the product is the probability of a word given all its predecessors. We can think of this as a *word prediction* task: given the context *Computers are*, we want to compute a probability over the next token. The relative frequency estimate of the probability of the word *useless* in this context is,

$$\begin{aligned}\text{p}(\textit{useless} \mid \textit{computers are}) &= \frac{\text{count}(\textit{computers are useless})}{\sum_{x \in \mathcal{V}} \text{count}(\textit{computers are } x)} \\ &= \frac{\text{count}(\textit{computers are useless})}{\text{count}(\textit{computers are})}.\end{aligned}$$

Note that we haven't made any approximations yet, and we could have just as well applied the chain rule in reverse order, $\text{p}(\boldsymbol{w}) = \text{p}(w_M) \times \text{p}(w_{M-1} \mid w_M) \times \ldots \times \text{p}(w_1 \mid w_2, \ldots, w_M)$, or in any other order. But this means that we also haven't really improved anything either: to compute the conditional probability $P(w_M \mid w_{M-1}, w_{M-2}, \ldots)$, we

[2] Chomsky has famously argued that this is evidence against the very concept of probabilistic language models: no such model could distinguish the grammatical sentence *colorless green ideas sleep furiously* from the ungrammatical permutation *furiously sleep ideas green colorless*. Indeed, even the bigrams in these two examples are unlikely to occur — at least, not in texts written before Chomsky proposed this example.

need to model $|\mathcal{V}|^{M-1}$ contexts. We cannot estimate such a distribution from any reasonable finite sample.

To solve this problem, n-gram models make a crucial simplifying approximation: condition on only the past $n-1$ words.

$$\text{p}(w_m \mid w_{m-1} \dots w_1) \approx \text{p}(w_m \mid w_{m-1}, \dots, w_{m-n+1}) \tag{5.7}$$

This means that the probability of a sentence $\boldsymbol{w}$ can be computed as

$$\text{p}(w_1, \dots, w_M) \approx \prod_m^M \text{p}(w_m \mid w_{m-1}, \dots, w_{m-n+1}) \tag{5.8}$$

To compute the probability of an entire sentence, it is convenient to pad the beginning and end with special symbols $\lozenge$ and $\blacklozenge$. Then the bigram ($n = 2$) approximation to the probability of *I like black coffee* is:

$$\begin{aligned}\text{p}(\textit{I like black coffee}) =& \text{p}(\textit{I} \mid \lozenge) \times \text{p}(\textit{like} \mid \textit{I}) \times \text{p}(\textit{black} \mid \textit{like}) \\ & \times \text{p}(\textit{coffee} \mid \textit{black}) \times \text{p}(\blacklozenge \mid \textit{coffee}).\end{aligned} \tag{5.9}$$

In this model, we have to estimate and store the probability of only $|\mathcal{V}|^n$ events, which is exponential in the order of the n-gram, and not $|\mathcal{V}|^M$, which is exponential in the length of the sentence. The n-gram probabilities can be computed by relative frequency estimation,

$$\Pr(W_m = i \mid W_{m-1} = j, W_{m-2} = k) = \frac{\text{count}(i, j, k)}{\sum_{i'} \text{count}(i', j, k)} = \frac{\text{count}(i, j, k)}{\text{count}(j, k)} \tag{5.10}$$

A key design question is how to set the hyperparameter n, which controls the size of the context used in each conditional probability. If this is misspecified, the language model will sacrifice accuracy.

When n is too small. Consider the following sentences:

(5.3) ***Gorillas*** *always like to groom* ***THEIR*** *friends.*

(5.4) *The* ***computer*** *that's on the 3rd floor of our office building* ***CRASHED****.*

The uppercase bolded words depend crucially on their predecessors in lowercase bold: the likelihood of *their* depends on knowing that *gorillas* is plural, and the likelihood of *crashed* depends on knowing that the subject is a *computer*. If the n-grams are not big enough to capture this context, then the resulting language model would offer probabilities that are too low for these sentences, and too high for sentences that fail basic linguistic tests like number agreement.

When n is too big. In this case, we cannot make good estimates of the n-gram parameters from our dataset, because of data sparsity. To handle the *gorilla* example, we would need to model 6-grams; which means accounting for $|\mathcal{V}|^6$ events. Under a very small vocabulary of $|\mathcal{V}| = 10^4$, this means estimating the probability of 10^{24} distinct events.

These two problems point to another **bias-variance** tradeoff. But in practice the situation is even worse: we often have **both** problems at the same time! Language is full of long-range dependencies that we cannot capture because n is too small; at the same time, language datasets are full of rare phenomena, whose probabilities we fail to estimate accurately because n is too large.

We will seek approaches to keep n large, while still making low-variance estimates of the underlying parameters. To do this, we will introduce a different sort of bias: **smoothing** . But first, let's take a digression to discuss how to evaluate language models.

5.2 Evaluating language models

Because language models are typically components of larger systems — language modeling is not usually an application itself — we would prefer **extrinsic evaluation**. This means evaluating whether the language model improves performance on the application task, such as machine translation or speech recognition. But this is often hard to do, and depends on details of the overall system which may be irrelevant to language modeling. In contrast, **intrinsic evaluation** is task-neutral. Better performance on intrinsic metrics may be expected to improve extrinsic metrics across a variety of tasks, unless we are overoptimizing the intrinsic metric. We will discuss intrinsic metrics here, but bear in mind that it is important to also perform extrinsic evaluations to ensure that the improvements obtained on these intrinsic metrics really carry over to the applications that we care about.

Held-out likelihood

A popular intrinsic metric is the **held-out likelihood**. To compute this metric, we "hold out" a portion of our data from training. We use the model estimated from the training set to compute the log probability of this held-out data, with the goal of assigning it high probability. Specifically, we compute,

$$\ell(\boldsymbol{w}) = \sum_{i}^{N} \sum_{m}^{M_i} \log \mathrm{p}(w_m^{(i)} \mid w_{m-1}^{(i)}, \ldots, w_{m-n+1}^{(i)}), \tag{5.11}$$

summing over all sentences $\{\boldsymbol{w}^{(i)}\}_{i \in 1 \ldots N}$ in the held-out dataset.

Typically, unknown words in the test data are mapped to the $\langle \text{UNK} \rangle$ token. This means that we have to estimate some probability for $\langle \text{UNK} \rangle$ on the training data. One way to do this is to fix the vocabulary $\mathcal{V}$ to the $|\mathcal{V}| - 1$ words with the highest counts in the training data, and then convert all other tokens to $\langle \text{UNK} \rangle$.

Perplexity

Perplexity is a transformation of the held-out likelihood into an information-theoretic quantity. Specifically, we compute

$$\text{Perplex}(\boldsymbol{w}) = 2^{-\frac{\ell(\boldsymbol{w})}{M}}, \tag{5.12}$$

where M is the total number of tokens in the held-out corpus.

Lower perplexities correspond to higher likelihoods, so lower scores are better on this metric. (How to remember: lower perplexity is better, because your language model is less perplexed.) To understand perplexity, here are some useful cases to consider:

- In the limit of a perfect language model, we would assign probability 1 to the held-out corpus, with $\text{Perplex}(\boldsymbol{w}) = 2^{-\log 1} = 2^0 = 1$.
- In the opposite limit, we assign the held-out corpus probability 0, which corresponds to an infinite perplexity.
- Assume a uniform, unigram model in which $\text{p}(w_i) = \frac{1}{|\mathcal{V}|}$ for all words in the vocabulary. Then,

$$\begin{aligned}\text{Perplex}(\boldsymbol{w}) &= \left[\left(\frac{1}{V}\right)^M\right]^{-\frac{1}{M}} \\ &= \left(\frac{1}{V}\right)^{-1} = V \end{aligned} \tag{5.13}$$

  This is the "worst reasonable case" scenario, since you could build such a language model without even looking at the data.

In practice, n-gram language models tend to give perplexities in the range between 1 and $|\mathcal{V}|$. For example, Jurafsky and Martin estimate a language model over a vocabularly of roughly $20,000$ words, on 38 million tokens of text from the Wall Street Journal (Jurafsky and Martin, 2009, page 97). They report the following perplexities on a held-out set of 1.5 million tokens:

- Unigram ($n = 1$): 962
- Bigram ($n = 1$): 170
- Trigram ($n = 3$): 109

As n increases, will the perplexity contrinute to decrease?

5.3 Smoothing and discounting

Limited data is a persistent problem in estimating language models. In section 5.1, we presented n-grams as a partial solution. But as we saw, sparse data can be a problem even for low-order n-grams; at the same time, many linguistic phenomena, like subject-verb agreement, cannot be incorporated into language models without higher-order n-grams. It is therefore necessary to add additional inductive biases to n-gram language models. This section covers some of the most intuitive and common approaches, but there are many more (Chen and Goodman, 1999).

Smoothing

A major concern in language modeling is to avoid the situation $\text{p}(\boldsymbol{w}) = 0$, which could arise as a result of a single unseen n-gram. A similar problem arose in Naïve Bayes, and there we solved it by **smoothing**: adding imaginary "pseudo" counts. The same idea can be applied to n-gram language models, as shown here in the bigram case,

$$\text{p}_{\text{smooth}}(w_m \mid w_{m-1}) = \frac{\text{count}(w_{m-1}, w_m) + \alpha}{\sum_{w' \in \mathcal{V}} \text{count}(w_m, w') + |\mathcal{V}|\alpha}. \tag{5.14}$$

This basic framework is called **Lidstone smoothing** , but special cases have other names:

- **Laplace smoothing** corresponds to the case $\alpha = 1$.
- **Jeffreys-Perks law** corresponds to the case $\alpha = 0.5$. Manning and Schütze (1999) offer more insight on the justifications for this setting.

To maintain normalization, anything that we add to the numerator (α) must also appear in the dominator ($|\mathcal{V}|\alpha$). This idea is reflected in the concept of **effective counts** :

$$c_i^* = (c_i + \alpha)\frac{M}{M + |\mathcal{V}|\alpha}, \tag{5.15}$$

where c_i is the count of event i, c_i^* is the effective count, and $M = \sum_i^{|\mathcal{V}|} c_i$ is the total number of terms in the dataset $(w_1, w_2, \ldots, w_M)$. This term ensures that $\sum^{|} \mathcal{V}|_i c_i^* = \sum^{|} \mathcal{V}|_i c_i = M$. The **discount** for each n-gram is then computed as,

$$d_i = \frac{c_i^*}{c_i} = \frac{(c_i + \alpha)}{c_i}\frac{M}{(M + \alpha)}$$

Discounting and backoff

Discounting "borrows" probability mass from observed n-grams and redistributes it. In Lidstone smoothing, we borrow probability mass by increasing the denominator of the

relative frequency estimates, and redistribute it by increasing the numerator for all n-grams. But instead, we could borrow the same amount of probability mass from all observed counts, and redistribute it among only the unobserved counts. This is called **absolute discounting.** For example, suppose we set an absolute discount $d = 0.1$ in a trigram model; the resulting counts and probabilities for the trigram context $(\text{denied}, \text{the}, \_)$ are shown in Table 5.1.

word	counts c	effective counts c^*	unsmoothed probability	smoothed probability
allegations	3	2.9	0.429	0.414
reports	2	1.9	0.286	0.271
claims	1	0.9	0.143	0.129
request	1	0.9	0.143	0.129
charges	0	0.2	0.000	0.029
benefits	0	0.2	0.000	0.029
...				

Table 5.1: Example of absolute discounting in a trigram language model, for the context $(\text{denied}, \text{the}, \_)$.

Discounting reserves some probability mass from the observed data; we need not redistribute this probability mass equally. Instead, we can **backoff** to a lower-order language model. In other words: if you have trigrams, use trigrams; if you don't have trigrams, use bigrams; if you don't even have bigrams, use unigrams. This is called **Katz backoff** :

$$c^*(i,j) = c(i,j) - d \tag{5.16}$$

$$\text{p}_{\text{Katz}}(i \mid j) = \begin{cases} \frac{c^*(i,j)}{c(j)} & \text{if } c(i,j) > 0 \\ \alpha(j) \times \frac{\text{p}_{\text{unigram}}(i)}{\sum_{i':c(i',j)=0} \text{p}_{\text{unigram}}(i')} & \text{if } c(i,j) = 0 \end{cases} \tag{5.17}$$

The term $\alpha(j)$ indicates the amount of probability mass that has been discounted for context j. This probability mass is then divided across all the unseen events, $\{i' : c(i', j) = 0\}$, proportional to the unigram probability of each word i'. The discount parameter d can be optimized to minimize perplexity on a development set.

Interpolation*

Backoff is one way to combine n-gram models across various values of n. An alternative approach is **interpolation** : setting the probability of a word in context to a weighted sum of its probabilities across progressively shorter contexts.

Instead of choosing a single n for the size of the n-gram, we can take the weighted average across several n-gram probabilities. For example, for an interpolated trigram

model,

$$\begin{aligned}\mathrm{p}_{\text{Interpolation}}(i \mid j,k) = &\lambda_3^{(i)}\mathrm{p}_3^*(i \mid j,k) \\ &+ \lambda_2^{(i)}\mathrm{p}_2^*(i \mid j) \\ &+ \lambda_3^{(i)}\mathrm{p}_1^*(i).\end{aligned}$$

In this equation, p_n^* is the maximum likelihood estimate (MLE) of an n-gram model, and $\lambda_n^{(i)}$ is the weight of the n-gram model p_n^* for word i. A nice property of this model is that it can learn to use longer context for some words (e.g., possessive pronouns like *his* and *her*, which often match the gender of the entity as defined earlier in the sentence), and shorter context for others (e.g., rare content words).

To ensure that the interpolated $\mathrm{p}(\boldsymbol{w})$ is still a probability, we have a constraint, $\sum_n \lambda_n^{(i)} = 1, \forall i$. But how to find the specific values of $\lambda^{(i)}$ for each word? An elegant solution is **expectation maximization** . Recall from chapter 4 that we can think about EM as learning with **missing data**: we just need to choose missing data such that learning would be easy if it weren't missing. What's missing in this case? We can think of each word w_m as drawn from an n-gram of unknown size, $z_m \in \{1 \dots n\}$. This z_m is the missing data that we are looking for.

Specifically, we apply the following generative story:

- For each token m,
  - draw $z_m \sim \text{Categorical}(\lambda^{(w_m)})$
  - draw $w_m \sim \mathrm{p}_{z_m}^*(w_m \mid w_{m-1}, \dots w_{m-z_m})$.

If we knew $\{z_m\}_{m \in 1 \dots M}$, then we could compute λ from relative frequency estimation,

$$\lambda_k^{(i)} = \frac{\text{count}(W_m = i, Z_m = k)}{\text{count}(W_m = i)}. \tag{5.18}$$

Since we do not know the values of the latent variables Z, we impute a distribution $q_m(z_m)$ in the E-step, which represents our degree of belief that word token w_m was generated from a n-gram of order z_m.

Having defined these quantities, we can derive EM updates:

- **E-step**:

$$q_m(n) = \Pr(Z_m = n \mid \boldsymbol{w}_{1:m}) \tag{5.19}$$

$$\propto \mathrm{p}_z^*(w_m \mid w_{m-1}, \dots, w_{m-n+1}) \times \lambda_n^{(w_m)} \tag{5.20}$$

- **M-step**:

$$\lambda_n(i) = \frac{E_q\left[\text{count}(W_m = i, Z_m = n)\right]}{\sum_{n'} E_q\left[\text{count}(W = i, Z = n')\right]} \tag{5.21}$$

$$= \frac{\sum_m q_m(n)\delta(W_m = i)}{\sum_m \delta(W_m = i)}, \tag{5.22}$$

where $\delta(w_m = i)$ is a delta function that takes the value 1 if the Boolean condition $w_m = i$ holds, and 0 otherwise. As usual, EM iterates between these two steps until convergence to a local optimum.

Kneser-Ney smoothing*

Kneser-Ney smoothing is based on absolute discounting, but it redistributes the resulting probability mass in a different way from Katz backoff. Empirical evidence points to Kneser-Ney smoothing as the state-of-art for n-gram language modeling **?**.

To motivate Kneser-Ney smoothing, consider the example: *I recently visited* _. Which of the following is more likely?

- *Francisco*
- *Duluth*

Now suppose that both bigrams *visited Duluth* and *visited Francisco* are unobserved in our training data, and furthermore, that the unigram probability $\text{p}^*(\textit{Francisco})$ is greater than $\text{p}^*(\textit{Duluth})$. Nonetheless we would still guess that $\text{p}(\textit{visited Duluth}) > P(\textit{visited Francisco})$, because *Duluth* is a more **versatile** word: it an occur in many contexts, while *Francisco* almost always occurs in a single context, following the word *San*. This notion of versatility is the key to Kneser-Ney smoothing.

Writing u for a context of undefined length, and $\text{count}(w, u)$ as the count of word w in context u, we define the Kneser-Ney bigram probability as

$$\text{p}_{KN}(w \mid u) = \begin{cases} \frac{\text{count}(w,u)-d}{\text{count}(u)}, & \text{count}(w, u) > 0 \\ \alpha(u) \times \text{p}_{\text{continuation}}(w), & \text{otherwise} \end{cases}$$

$$\text{p}_{\text{continuation}}(w) = \frac{|u : \text{count}(w, u) > 0|}{\sum_{w' \in \mathcal{V}} |u' : \text{count}(w', u') > 0|}.$$

First, note that we reserve probability mass using absolute discounting d, which is taken from all unobserved n-grams. The total amount of discounting in context u is $d \times |w : \text{count}(w, u) > 0|$, and we divide this probability mass equally among the unseen n-grams,

$$\alpha(u) = |w : \text{count}(w, u) > 0| \times \frac{d}{\text{count}(u)}. \tag{5.23}$$

This is the amount of probability mass left to account for versatitility, which we define via the *continuation probability* $\text{p}_{\text{continuation}}(w)$ as proportional to the number of observed contexts in which w appears. In the numerator of the continuation probability we have the number of contexts u in which w appears, and in the denominator, we normalize by summing the same quantity over all words w'.

The idea of modeling versatility by counting contexts may seem heuristic, but there is an elegant theoretical justification from Bayesian nonparametrics (Teh, 2006). Kneser-Ney smoothing on n-grams was the dominant language modeling technique — widely used in speech recognition and machine translation — before the arrival of neural language models.

5.4 Recurrent neural network language models

Until around 2010, n-grams were the universal solution to language modeling problems. But in a few years, they have been almost completely supplanted by a new family of approaches based on **neural networks** . These models do not make the n-gram assumption of restricted context; indeed, they can incorporate arbitrarily distant contextual information, while remaining computationally and statistically tractable.

The first insight is to treat word prediction as a **discriminative** learning task: rather than directly estimating the distribution $\text{p}(w \mid u)$ from (smoothed) relative frequencies, we now treat language modeling as a machine learning problem, and estimate parameters that maximize the log conditional probability of a corpus.[3]

The second insight is to reparametrize the probability distribution $\text{p}(w \mid u)$ as a function of two dense K-dimensional numerical vectors, $\boldsymbol{\beta}_w \in \mathbb{R}^K$, and $\boldsymbol{v}_u \in \mathbb{R}^K$,

$$\text{p}(w \mid u) = \frac{\exp(\boldsymbol{\beta}_w \cdot \boldsymbol{v}_u)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{v}_u)}, \tag{5.24}$$

where $\boldsymbol{\beta}_w \cdot \boldsymbol{v}_c$ represents a dot product. In the neural networks literature, this function is sometimes known as a **softmax** layer. Note that the denominator ensures that it is a properly normalized probability distribution.

The word vectors $\boldsymbol{\beta}_w$ are parameters of the model, and are estimated directly. As we will see in chapter 15, these vectors carry useful information about word meaning, and semantically similar words tend to have highly correlated vectors.

The context vectors $\boldsymbol{v}_u$ can be computed in various ways, depending on the model. Here we will consider a relatively simple — but effective — neural language model, the **recurrent neural network** (RNN Mikolov et al., 2010). The basic idea is to recurrently update the context vectors as we move through the sequence. Let us write $\boldsymbol{h}_m$ for the

[3]This idea is not in itself new; for example, Rosenfeld (1996) applies logistic regression to language modeling, and Roark et al. (2007) apply perceptrons and conditional random fields (section 6.5).

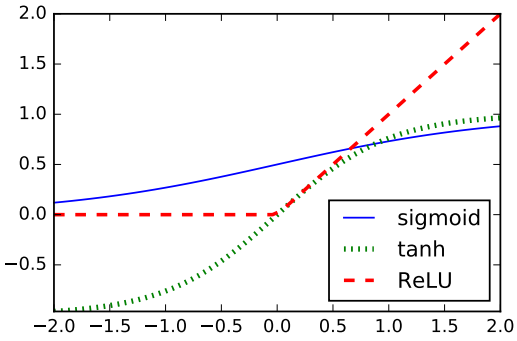

Figure 5.1: Nonlinear activation functions for neural networks

[todo: make figure]

Figure 5.2: The recurrent neural network, viewed as a computation graph. Solid lines indicate probabilistic dependencies, dashed red lines indicate direct computations, and dotted blue lines indicate backpropagation.

contextual information at position m in the sequence. RNNs employ the following recurrence:

$$\boldsymbol{x}_m \triangleq \boldsymbol{\phi}_{w_m} \tag{5.25}$$

$$\boldsymbol{h}_m = g(\Theta \boldsymbol{h}_{m-1} + \boldsymbol{x}_m) \tag{5.26}$$

$$\text{p}(w_{m+1} \mid w_1, w_2, \ldots, w_m) = \frac{\exp(\boldsymbol{\beta}_{w_{m+1}} \cdot \boldsymbol{h}_m)}{\sum_{w' \in \mathcal{V}} \exp(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_m)}, \tag{5.27}$$

where $\boldsymbol{\phi}$ is a matrix of **input word embeddings**, and $\boldsymbol{x}_m$ denotes the embedding for word w_m. The function g is an element-wise nonlinear **activation function**. Typical choices are:

- $\tanh(x)$, the hyperbolic tangent;
- $\sigma(x)$, the **sigmoid function** $\frac{1}{1+\exp(-x)}$;
- $(x)_+$, the **rectified linear unit**, $(x)_+ = \max(x, 0)$, also called **ReLU**.

These activation functions are shown in Figure 5.1. The sigmoid and tanh functions "squash" their inputs into a fixed range: $[0, 1]$ for the sigmoid, $[-1, 1]$ for $\tanh$. This makes it possible to chain together many iterations of these functions without numerical instability.

A key point about the RNN language model is that although each w_m depends only on the context vector $\boldsymbol{h}_{m-1}$, this vector is in turn influenced by **all** previous tokens, $w_1, w_2, \ldots w_{m-1}$,

through the recurrence operation: w_1 affects $\boldsymbol{h}_1$, which affects $\boldsymbol{h}_2$, and so on, until the information is propagated all the way to $\boldsymbol{h}_m$ (see Figure 5.2). This is an important distinction from n-gram language models, where any information outside the n-word window is ignored. Thus, in principle, the RNN language model can handle long-range dependencies, such as number agreement over long spans of text — although it would be difficult to know where exactly in the vector $\boldsymbol{h}_m$ this information is represented. The main limitation is that information is attenuated by repeated application of the nonlinearity g, particularly when a "squashing function" such as the sigmoid or $\tanh$ are used. **Long short-term memories** (LSTMs), described below, are a variant of RNNs that address this issue, using memory cells to propagate information through the sequence without applying non-linearities (Hochreiter and Schmidhuber, 1997).

The computation of the denominator in Equation 5.27 is a bottleneck, because it involves summing over the entire vocabulary. One solution is to use a **hierarchical softmax** function, which computes the sum more efficiently by organizing the vocabulary into a tree (Mikolov et al., 2011). Another strategy is to optimize an alternative metric, such as noise-contrastive estimation (Gutmann and Hyvärinen, 2012), which learns by distinguishing observed instances from artificial instances generated from a noise distribution (Mnih and Teh, 2012).

Estimation by backpropagation

The recurrent neural network language model has the following parameters:

- $\boldsymbol{\phi}_i \in \mathbb{R}^K$, the "input" word vectors (these are sometimes called **word embeddings**, since each word is embedded in a K-dimensional space);
- $\boldsymbol{\beta}_i \in \mathbb{R}^K$, the "output" word vectors;
- $\Theta \in \mathbb{R}^{K \times K}$, the recurrence operator.

Each of these parameters must be estimated. We do this by formulating an objective function over the training corpus, $\ell(\boldsymbol{w})$, and then employ **backpropagation** to incrementally update the parameters after encountering each training example. Backpropagation is a term from the neural network literature, which means that we use the chain rule of differentiation to obtain gradients on each parameter. For example, suppose we want to obtain the gradient of the log-likelihood with respect to a single row of the recurrence operator, $\boldsymbol{\theta}_k$. Let us define the local error function e_m,

$$e_m(\boldsymbol{h}_m) \triangleq -\log \mathrm{p}(w_m \mid w_1, w_2, \ldots, w_{m-1}) \tag{5.28}$$

$$= -\boldsymbol{\beta}_{w_m} \cdot \boldsymbol{h}_{m-1} + \log \sum_{w'} \exp\left(\boldsymbol{\beta}_{w'} \cdot \boldsymbol{h}_{m-1}\right) \tag{5.29}$$

$$\ell(\boldsymbol{w}) = \sum_{m}^{M} e_m(\boldsymbol{h}_{m-1}), \tag{5.30}$$

so the total error is the sum of the local errors.

We can now differentiate with respect to $\boldsymbol{\theta}_k$:

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} \ell(\boldsymbol{w}) = \sum_m \frac{\partial e_m(\boldsymbol{h}_{m-1})}{\partial \boldsymbol{\theta}_k} \tag{5.31}$$

$$= \sum_m e'_m(\boldsymbol{h}_m) \frac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{h}_{m-1}. \tag{5.32}$$

In the first line, we simply distribute the derivative across the sum. In the second line, we apply the chain rule of calculus. The term $e'_m(\boldsymbol{h}_m)$ refers to the gradient of the function e_m evaluated at $\boldsymbol{h}_m$, which is a scalar. Next we compute the derivative of $\boldsymbol{h}_{m-1}$, first noting that within the vector $\boldsymbol{h}_{m-1}$, only the element $h_{m-1,k}$ depends on $\boldsymbol{\theta}_k$.

$$\frac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{h}_{m-1} = \frac{\partial}{\partial \boldsymbol{\theta}_k} h_{m-1,k} \tag{5.33}$$

$$= g(\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}) \tag{5.34}$$

$$= g'(\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}) \frac{\partial}{\partial \boldsymbol{\theta}_k} (\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}) \tag{5.35}$$

$$= g'(\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}) \times (\boldsymbol{h}_{m-2} + \boldsymbol{\theta}_k \odot \frac{\partial}{\partial \boldsymbol{\theta}_k} \boldsymbol{h}_{m-2}), \tag{5.36}$$

where g' is the gradient of the elementwise nonlinearity function, evaluated at $\boldsymbol{\theta}_k \cdot \boldsymbol{h}_{m-2} + x_{m-1,k}$, and $\odot$ is an elementwise (Hadamard) vector product.

A key point is that the derivative $\frac{\partial \boldsymbol{h}_{m-1}}{\partial \boldsymbol{\theta}_k}$ depends recursively on $\frac{\partial \boldsymbol{h}_{m-2}}{\partial \boldsymbol{\theta}_k}$. Furthermore, we will need to compute $\frac{\partial \boldsymbol{h}_{m-2}}{\partial \boldsymbol{\theta}_k}$ **again**, to account for the error term $e_{m-1}(\boldsymbol{h}_{m-2})$. To avoid redoing work, it is best to compute and cache all necessary gradients during the forward pass, so that they can be reused during backpropagation. This idea is implemented by neural network toolkits such as TensorFlow (Abadi et al., 2016) and Torch (Collobert et al., 2011). To use these toolkits, the user defines a **computation graph** representing the neural network structure, which culminates in a scalar loss function. The toolkit then automatically computes the gradient of the loss function with respect to all model parameters. Optimization is usually performed using an online method such as stochastic gradient descent (section 2.4). It is important to note that the log-likelihood of an RNNLM is a non-convex function of the parameters, so there is no learning procedure that is guaranteed to converge to the global optimum. [todo: explicitly define backpropagation algorithm]

Hyperparameters

The RNN language model has several hyperparameters that must be tuned to ensure good performance. The model capacity is controlled by the size of the word and context vectors K, which play a role that is somewhat analogous to the size of the n-gram context.

For datasets that are large with respect to the vocabulary (i.e., there is a large token-to-type ratio), we can afford to estimate a model with a large K, which enables more subtle distinctions between words and contexts. When the dataset is relatively small, then K must be smaller too. However, this general advice has not yet been formalized into any concrete formula for choosing K, and trial-and-error is still necessary. Overfitting can also be prevented by **dropout** , which involves randomly setting some elements of the computation to zero (Srivastava et al., 2014), which can force the learner not to overly rely on any particular dimension of the word or context vectors. (The dropout rate must be tuned by the user.) Other design decisions include: the nature of the nonlinear activation function g, the size of the vocabulary to try to model, and the parametrization of the learning algorithm, such as the learning rate.

Alternative neural language models

A well known problem with RNNs is that backpropagation across long chains tends to lead to "vanishing" or "exploding" gradients (Bengio et al., 1994). For example, the input embedding of word w_1 affects the likelihood of a distance word such as w_9, but this impact may be attenuated by backpropagation through the eight intervening time steps. One solution is to rescale the gradients, or to clip them at some maximum value (Pascanu et al., 2013).

A popular variant of RNNs, which is more robust to these problems, is the **long short-term memory** (**LSTM** ; Hochreiter and Schmidhuber, 1997; Sundermeyer et al., 2012). This model augments the hidden state $\boldsymbol{h}_m$ with a "memory cell" $\boldsymbol{c}_m$. The value of the memory cell at each time m is a linear interpolation between its previous value $\boldsymbol{c}_{m-1}$, and an update that is computed from the current input $\boldsymbol{x}_m$ and the previous hidden state $\boldsymbol{h}_{m-1}$. The next hidden state $\boldsymbol{h}_m$ is then computed from the memory cell. The interpolation weights are controlled by a set of gates, which are themselves functions of the input and previous hidden state. The gates are computed from sigmoid activations, ensuring that their values will be in the range $[0, 1]$; they thus simulate logical gates, but are differentiable. The complete LSTM update equations are:

$$\boldsymbol{f}_m = \sigma(\Theta^{(h\to f)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(x\to f)} \cdot \boldsymbol{x}_m) \quad \text{forget gate} \quad (5.37)$$

$$\boldsymbol{i}_m = \sigma(\Theta^{(h\to i)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(x\to i)} \cdot \boldsymbol{x}_m) \quad \text{input gate} \quad (5.38)$$

$$\tilde{\boldsymbol{c}}_m = \tanh(\Theta^{(h\to c)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(w\to c)} \cdot \boldsymbol{x}_m) \quad \text{update candidate} \quad (5.39)$$

$$\boldsymbol{c}_m = \boldsymbol{f}_m \odot \boldsymbol{c}_{m-1} + \boldsymbol{i}_m \odot \tilde{\boldsymbol{c}}_m \quad \text{memory cell update} \quad (5.40)$$

$$\boldsymbol{o}_m = \sigma(\Theta^{(h\to o)} \cdot \boldsymbol{h}_{m-1} + \Theta^{(x\to o)} \cdot \boldsymbol{x}_m) \quad \text{output gate} \quad (5.41)$$

$$\boldsymbol{h}_m = \boldsymbol{o}_m \odot \boldsymbol{c}_m \quad \text{output.} \quad (5.42)$$

In these equations, $\odot$ refers to an elementwise product. The LSTM model has been shown to outperform RNNs across a wide range of problems (it was first used for language modeling by Sundermeyer et al. (2012)), and is now widely used for sequence

modeling tasks. There are several LSTM variants, of which the Gated Recurrent Unit (Cho et al., 2014) is presently one of the more well known. Many software packages implement recurrent neural networks, so choosing between these various architectures is simple from a user's perspective. Jozefowicz et al. (2015) provide an empirical comparison of various modeling choices circa 2015. Notable earlier non-recurrent architectures include the neural probabilistic language model (Bengio et al., 2003) and the log-bilinear language model (Mnih and Hinton, 2007).

5.5 Out-of-vocabulary words

Through this chapter, we have assumed a **closed-vocabulary** setting — the vocabulary $\mathcal{V}$ is assumed to be a finite set. In realistic application scenarios, this assumption may not hold. Consider, for example, the problem of translating newspaper articles. The following sentence appeared in a Reuters article on January 6, 2017:[4]

> The report said U.S. intelligence agencies believe Russian military intelligence, the **GRU**, used intermediaries such as **WikiLeaks**, **DCLeaks.com** and the **Guccifer** 2.0 "persona" to release emails...

Language models are often trained on the Gigaword corpus[5], which was released in 2003. The bolded terms either did not exist at this date, or were not widely known; they are unlikely to be in the vocabulary. The same problem can occur for a variety of other terms: new technologies, previously unknown individuals, new words (e.g., *hashtag*), and numbers.

One solution is to simply mark all such terms with a special token, $\langle \text{UNK} \rangle$. While training the language model, we decide in advance on the vocabulary (often the K most common terms), and mark all other terms in the training data as $\langle \text{UNK} \rangle$. If we do not want to determine the vocabulary size in advance, an alternative approach is to simply mark the first occurrence of each word type as $\langle \text{UNK} \rangle$.

In some scenarios, we may prefer to make distinctions about the likelihood of various unknown words. This is particularly important in languages that have rich morphological systems, with many inflections for each word. For example, Spanish is only moderately complex from a morphological perspective, yet each verb has dozens of inflected forms. In such languages, there will necessarily be many word types that we do not encounter in a corpus, which are nonetheless predictable from the morphological rules of the language. To use a somewhat contrived English example, if *transfenestrate* is in the vocabulary, our

[4] Bayoumy, Y. and Strobel, W. (2017, January 6). U.S. intel report: Putin directed cyber campaign to help Trump. *Reuters*. Retrieved from `http://www.reuters.com/article/us-usa-russia-cyber-idUSKBN14Q1T8` on January 7, 2017.

[5] `https://catalog.ldc.upenn.edu/LDC2003T05`

language model should be able to assign a probability to the past tense *transfenestrated* even if it does not appear in the training data.

One way to accomplish this is to supplement word-level language models with **character-level language models** . Such models can use n-grams or RNNs, but with a fixed vocabulary equal to the set of ASCII or Unicode characters. For example Ling et al. (2015) propose an LSTM model over characters, and Kim (2014) employ a **convolutional neural network** (LeCun and Bengio, 1995). A more linguistically motivated approach is to segment words into meaningful subword units, known as **morphemes** (see chapter 9). For example, Botha and Blunsom (2014) induce vector representations for morphemes, which they build into a log-bilinear language model; Bhatia et al. (2016) incorporate morpheme vectors into an LSTM.

Part II

Sequences and trees

Chapter 6

Sequence labeling

In sequence labeling, we want to assign tags to words, or more generally, we want to assign discrete labels to elements in a sequence. There are many applications of sequence labeling in natural language processing, and chapter 7 presents an overview. One of the most classic application of sequence labeling is **part-of-speech tagging** , which involves tagging each word by its grammatical category. Coarse-grained grammatical categories include **N**OUNs, which describe things, properties, or ideas, and **V**ERBs, which describe actions and events. Given a simple sentence like,

(6.1) They can fish.

we would like to produce the tag sequence N V V, with the modal verb *can* labeled as a verb in this simplified example.

6.1 Sequence labeling as classification

One way to solve tagging problems is to treat them as classification. A simple tagging model would have a single base feature, the word itself:

$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \text{N}, 1) = \langle \textit{they}, \text{N} \rangle \tag{6.1}$$
$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \text{V}, 2) = \langle \textit{can}, \text{V} \rangle \tag{6.2}$$
$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \text{V}, 3) = \langle \textit{fish}, \text{V} \rangle. \tag{6.3}$$

Here the feature function takes three arguments as input: the sentence to be tagged (*they can fish* in all cases), the proposed tag (e.g., N or V), and the word token to which this tag is applied. This simple feature function then returns a single feature: a tuple including the word to be tagged and the tag that has been proposed. If the vocabulary size is V and the number of tags is K, then there are $V \times K$ features. Each of these features must be assigned a weight. These weights can be learned from a labeled dataset using a classification algorithm such as perceptron, but this isn't necessary in this case: it would be

equivalent to define the classification weights directly from a dictionary, with $\theta_{w,y} = 1$ for the best tag y for each word w, and $\theta_{w,y} = 0$ for all other tags.

However, it is easy to see that this simple classification approach can go wrong. Consider the word *fish*, which often describes the animal rather than the activity; in these cases, *fish* should be tagged as a noun. To tag ambiguous words correctly, the tagger must rely on context, such as the surrounding words. We can build this context into the feature set by incorporating the surrounding words as additional features:

$$\begin{aligned}
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \text{N}, 1) = \{&\langle w_i = \textit{they}, y_i = \text{N}\rangle, \\
&\langle w_{i-1} = \lozenge, y_i = \text{N}\rangle, \\
&\langle w_{i+1} = \textit{can}, y_i = \text{N}\rangle\} \qquad (6.4)\\
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \text{V}, 2) = \{&\langle w_i = \textit{can}, y_i = \text{V}\rangle, \\
&\langle w_{i-1} = \textit{they}, y_i = \text{V}\rangle, \\
&\langle w_{i+1} = \textit{fish}, y_i = \text{V}\rangle\} \qquad (6.5)\\
\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \text{V}, 3) = \{&\langle w_i = \textit{fish}, y_i = \text{V}\rangle, \\
&\langle w_{i-1} = \textit{can}, y_i = \text{V}\rangle, \\
&\langle w_{i+1} = \blacklozenge, y_i = \text{V}\rangle\}. \qquad (6.6)
\end{aligned}$$

These features contain enough information that a tagger should be able to choose the right label for the word *fish*: words that follow the modal verb *can* are likely to be verbs themselves, so the feature $\langle w_{i-1} = \textit{can}, y_i = \text{V}\rangle$ should have a large positive weight.

However, even with this enhanced feature set, it may be difficult to tag some sequences correctly. One reason is that there are often relationships between the tags themselves. For example, in English it is relatively rare for a verb to follow another verb — particularly if we differentiate **M**ODAL verbs like *can* and *should* from more typical verbs, like *give*, *transcend*, and *befuddle*. We would like to incorporate preferences **against** such tag sequences, and preferences **for** other tag sequences, such as NOUN-VERB.

The need for such preferences is best illustrated by a **garden path sentence** :

(6.2) The old man the boat.

Grammatically, the word *the* is a **D**ETERMINER. When you read the sentence, what part of speech did you first assign to *old*? Typically, this word is an ADJECTIVE — abbreviated as J — which is a class of words that modify nouns. Similarly, *man* is usually a noun. The resulting sequence of tags is D J N D N. But this is a mistaken "garden path" interpretation, which ends up leading nowhere. It is unlikely that a determiner would directly follow a noun,[1] and particularly unlikely that the entire sentence would lack a verb. The only possible verb in the sentence is the word *man*, which can refer to the act of maintaining and piloting something — often boats. But if *man* is tagged as a verb, then *old* is seated between a determiner and a verb, and must be a noun. And indeed, adjectives can often have a second interpretation as nouns when used in this way (e.g., *the young*, *the restless*).

[1]The main exception is the double object construction, as in *I gave the child a toy.*

This reasoning, in which the labeling decisions are intertwined, cannot be applied in a setting where each tag is produced by an independent classification decision.

6.2 Sequence labeling as structure prediction

As an alternative, we can think of the entire sequence of tags as a label itself. For a given sequence of words $\boldsymbol{w}_{1:M} = (w_1, w_2, \ldots, w_M)$, there is a set of possible taggings $\mathcal{Y}(\boldsymbol{w}_{1:M}) = \mathcal{Y}^M$, where $\mathcal{Y} = \{\text{N}, \text{V}, \text{D}, \ldots\}$ refers to the set of individual tags, and $\mathcal{Y}^M$ refers to the set of tag sequences of length M. We can then treat the sequence labeling problem as a classification problem in this exponential label space,

$$\hat{\boldsymbol{y}}_{1:M} = \underset{\boldsymbol{y}_{1:M} \in \mathcal{Y}(\boldsymbol{w}_{1:M})}{\text{argmax}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}_{1:M}, \boldsymbol{y}_{1:M}), \tag{6.7}$$

where $\boldsymbol{y}_{1:M} = (y_1, y_2, \ldots, y_M)$ is a sequence of M tags. Note that in this formulation, we have a feature function that consider the entire tag sequence $\boldsymbol{y}_{1:M}$. Such a feature function can therefore include features that capture the relationships between tagging decisions, such as the preference that determiners not follow nouns, or that all sentences have verbs.

Is it possible to perform tagging in this way? The problem of making a series of interconnected labeling decisions is known as **inference** . Because natural language is full of interrelated grammatical structures, inference is a crucial aspect of contemporary natural language processing. Can we perform inference over label sequences? In English, it is not unusual to have sentences of length $M = 20$; part-of-speech tag sets vary in size from 10 to several hundred. Taking the low end of this range, we have $\#|\mathcal{Y}(\boldsymbol{w}_{1:M})| \approx 10^{20}$, one hundred billion billion possible tag sequences. Enumerating and scoring each of these sequences would require an amount of work that is exponential in the sequence length; in other words, inference is intractable.

However, the situation changes when we restrict the feature function. Suppose we choose features that never consider more than one tag. We can indicate this restriction as,

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, m), \tag{6.8}$$

meaning that the overall feature vector is the sum of feature vectors associated with individual tagging decisions. Such features are not capable of capturing the intuitions that might help us solve garden path sentences, such as the insight that determiners rarely follow nouns in English. But this restriction does make it possible to find the globally

optimal tagging, by making a sequence of individual tagging decisions.

$$\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \boldsymbol{\theta}^\top \sum_m \boldsymbol{f}(\boldsymbol{w}, y_m, m) \tag{6.9}$$

$$= \sum_m \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, m) \tag{6.10}$$

$$\hat{y}_m = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, m) \tag{6.11}$$

$$\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_M) \tag{6.12}$$

Note that we are still searching over an exponentially large set of tag sequences! But the feature set restriction results in decoupling the labeling decisions that were previous interconnected. As a result, it is not necessary to score every one of the M^K tag sequences individually — we can find the optimal sequence by scoring the local parts of these decisions.

Now let's consider a slightly less restrictive feature function: rather than considering only individual tags, we will consider adjacent tags too. This means that we can have negative weights for infelicitous tag pairs, such as noun-determiner, and positive weights for typical tag pairs, such as determiner-noun and noun-verb. We define this feature function as,

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \tag{6.13}$$

Let's apply this feature function to the shorter example, *they can fish*, using features for word-tag and tag-tag pairs:

$$\boldsymbol{f}(\boldsymbol{w} = \textit{they can fish}, \boldsymbol{y} = \text{N V V}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.14}$$

$$= \boldsymbol{f}(\boldsymbol{w}, \text{N}, \lozenge, 1) + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{N}, 2) + \boldsymbol{f}(\boldsymbol{w}, \text{V}, \text{V}, 3) \tag{6.15}$$

$$= \langle w_m = \textit{they}, y_m = \text{N} \rangle + \langle y_m = \text{N}, y_{m-1} = \lozenge \rangle + \langle w_m = \textit{can}, y_m = \text{V} \rangle + \langle y_m = \text{V}, y_{m-1} = \text{N} \rangle + \langle w_m = \textit{fish}, y_m = \text{V} \rangle + \langle y_m = \text{V}, y_{m-1} = \text{V} \rangle + \langle y_m = \blacklozenge, y_{m-1} = \text{V} \rangle \tag{6.16}$$

We end up with seven active features: one for each word-tag pair, and one for each tag-tag pair (this includes a final tag $y_{M+1} = \blacklozenge$). These features capture what are arguably the two main sources of information for part-of-speech tagging: which tags are appropriate for each word, and which tags tend to follow each other in sequence. Given appropriate

weights for these features, we can expect to make the right tagging decisions, even for difficult cases like *the old man the boat*.

The example shows that even with the restriction to the feature set shown in Equation 6.13, it is still possible to construct expressive features that are capable of solving many sequence labeling problems. But the key question is: does this restriction make it possible to perform efficient inference? The answer is yes, and the solution is the Viterbi algorithm Viterbi (1967).

6.3 The Viterbi algorithm

We now consider the inference problem,

$$\hat{\boldsymbol{y}} = \operatorname*{argmax}_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.17}$$

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \tag{6.18}$$

Given this restriction on the feature function, we can solve this inference problem using **dynamic programming** , a algorithmic technique for reusing work in recurrent computations. As in many dynamic programming problems, we begin by solving an auxiliary problem: rather than finding the best tag sequence, we simply try to compute the **score** of the best tag sequence,

$$\max_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \max_{\boldsymbol{y}_{1:M}} \sum_{m=1}^{M} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.19}$$

$$= \max_{\boldsymbol{y}_{1:M}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M) + \sum_{m=1}^{M-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.20}$$

$$= \max_{y_M} \max_{y_{M-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M) + \max_{\boldsymbol{y}_{1:M-2}} \sum_{m=1}^{M-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m). \tag{6.21}$$

In this derivation, we first removed the final element $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_M, y_{M-1}, M)$ from the sum over the sequence, and then we adjusted the scope of the the $\max$ operation, since the elements $(y_1 \dots y_{M-2})$ are irrelevant to the final term.

Let us now define the **Viterbi variable** ,

$$v_m(y) \triangleq \max_{\boldsymbol{y}_{1:m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n), \tag{6.22}$$

where lower-case m indicates any position in the sequence, and y indicates a tag for that position. This variable represents the score of the best tag sequence $(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ that terminates in $\hat{y}_m = y$. From this definition, we can compute the score of the best tagging of the sequence by plugging the Viterbi variables $v_M(\cdot)$ into Equation 6.21,

$$\max_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \max_k v_M(k). \tag{6.23}$$

Now, let us look more closely at how we can compute these Viterbi variables.

$$\begin{aligned}
v_m(y) &\triangleq \max_{\boldsymbol{y}_{1:m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) && (6.24)\\
&= \max_{y_{m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) \\
&\quad + \max_{\boldsymbol{y}_{1:m-2}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_{m-1}, y_{m-2}) + \sum_{n=1}^{m-2} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) && (6.25)\\
&= \max_{y_{m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + v_{m-1}(y_{m-1}) && (6.26)\\
v_1(y) &= \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, \diamondsuit, 1). && (6.27)
\end{aligned}$$

Equation 6.26 is a **recurrence** for computing the Viterbi variables: each $v_m(y)$ can be computed in terms of $v_{m-1}(\cdot)$, and so on. We can therefore step forward through the sequence, computing first all variables $v_1(\cdot)$ from Equation 6.27, and then computing all variables $v_2(\cdot)$, $v_3(\cdot)$, and so on, until we reach the final set of variables $v_M(\cdot)$.

Graphically, it is customary to arrange these variables graphically with the sequence index m on the rows, and the tag index y on the columns; in this representation, each $v_{m-1}(y)$ is connected to each $v_m(y)$, forming a **trellis** . This is shown in Figure 6.1. As shown in the figure, we set aside special nodes for the start and end states.

Our real goal is to find the best scoring sequence, not simply to compute its score. But as is often the case in dynamic programming, solving the auxiliary program gets us almost all the way to our original goal. Recall that each $v_m(k)$ represents the score of the best tag sequence ending in $Y_m = k$. To compute this, we maximize over possible values of Y_{m-1}. If we keep track of the tag that maximizes this choice at each step, then we can walk backwards from the final tag, and recover the optimal tag sequence. This is indicated in Figure 6.1 by the solid blue lines, which we trace back from the final position.

Why does this work? We can make an inductive argument. Suppose k is indeed the optimal tag for word m, and we now need to decide on the tag Y_{m-1}. Because we make the inductive assumption that we know $Y_m = k$, and because the feature function is restricted to adjacent tags, we need not consider any of the tags $\boldsymbol{Y}_{m+1:M}$; these tags, and the features that describe them, are irrelevant to the inference of Y_{m-1}, given the choice of $Y_m = k$.

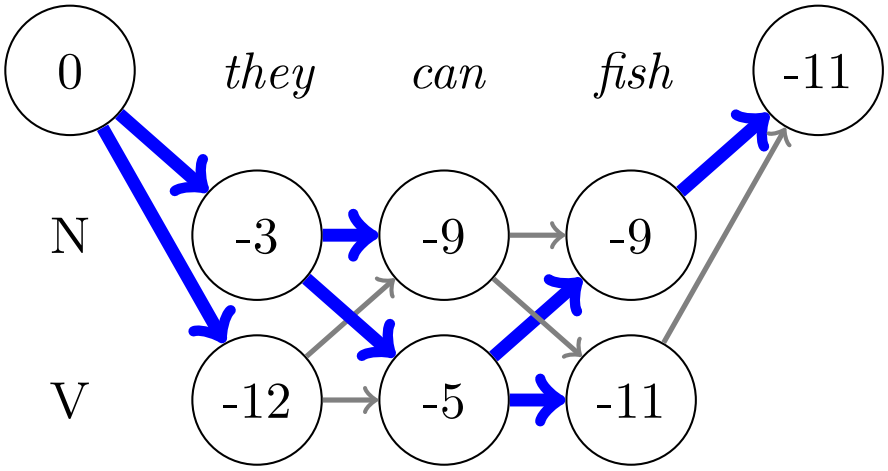

Figure 6.1: The trellis representation of the Viterbi variables, for the example *They can fish.*

Algorithm 5 The Viterbi algorithm.

for $k \in \{0, \dots K\}$ **do**
  $v_1(k) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, \lozenge, m)$
for $m \in \{2, \dots, M\}$ **do**
  for $k \in \{0, \dots, K\}$ **do**
    $v_m(k) = \max_{k'} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, k', m) + v_{m-1}(k')$
    $b_m(k) = \text{argmax}_{k'} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, k', m) + v_{m-1}(k')$
$y_M = \text{argmax}_k \, v_M(k) + \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \blacklozenge, k, M+1)$
for $m \in \{M-1, \dots 1\}$ **do**
  $y_m = b_m(y_{m+1})$

Thus, we are looking for the tag $\hat{y}_{m-1}$ that maximizes,

$$\hat{y}_{m-1} = \underset{y_{m-1}}{\text{argmax}} \, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, Y_m = k, Y_{m-1} = y_{m-1}, m) + \max_{\boldsymbol{y}_{1:m-2}} \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) \tag{6.28}$$

$$= \underset{y_{m-1}}{\text{argmax}} \, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, Y_m = k, y_{m-1}, m) + v_{m-1}(y_{m-1}), \tag{6.29}$$

which we obtain by plugging in the definition of the Viterbi variable.

The complete Viterbi algorithm is shown in Algorithm 5. This formalizes the recurrences that were described in the previous two paragraphs, and handles the boundary conditions at the start and end of the sequence. Specifically, when computing the initial Viterbi variables $v_1(\cdot)$, we use a special tag, $\lozenge$, to indicate the start of the sequence.

	they	*can*	*fish*
N	-2	-3	-3
V	-10	-1	-3

(a) Weights for "emission" features.

	N	V	♦
◊	-1	-2	$-\infty$
N	-3	-1	-2
V	-1	-3	-2

(b) Weights for "transition" features.

Table 6.1: Feature weights for the example trellis shown in Figure 6.1. Emission weights from ◊ and ♦ are implicitly set to $-\infty$. [todo: double check that this example works.]

When computing the final tag Y_M, we use another special tag, ♦, to indicate the end of the sequence. These special tags enable the use of transition features for the tags that begin and end the sequence: for example, nouns are relatively likely to start part-of-speech sequences in English, so we would like to have a positive weight for the feature $\langle \diamondsuit, \text{N} \rangle$; conjunctions are unlikely to end sequences, so we would like a negative weight for the feature $\langle \text{CC}, \blacklozenge \rangle$.

What is the complexity of this algorithm? If there are K tags and M positions in the sequence, then there are $M \times K$ Viterbi variables to compute. To compute each variable, we must compute a maximum over K possible predecessor tags. The total computation cost of populating the trellis is therefore $\mathcal{O}(MK^2)$, with an additional factor for the number of active features at each position. After completing the trellis, we simply trace the backwards pointers to the beginning of the sequence, which takes $\mathcal{O}(M)$ operations.

Example

We now illustrate the Viterbi algorithm with an example, using the minimal tagset $\{\text{N}, \text{V}\}$, corresponding to nouns and verbs. Even in this tagset, there is considerable ambiguity: consider the words *can* and *fish*, which can each take both tags. The sentence *They can fish* therefore has four possible taggings, two of which are grammatical. The tagging *They*/N *can*/V *fish*/N corresponds to the scenario of putting fish into cans.)

To begin, we use the feature weights defined in Table 6.1. These weights are used to incrementally fill in the trellis. As described in Algorithm 5, we fill in the cells from left to right, with each column corresponding to a word in the sequence. As we fill in the cells, we must keep track of the "back-pointers" $b_m(k)$ — the previous cell that maximizes the score of tag k at word m. These are represented in the figure with the thick blue lines. At the end of the algorithm, we recover the optimal tag sequence by tracing back the optimal path from the final position, $(M+1, k = \blacklozenge)$.

Higher-order features

The Viterbi algorithm was made possible by a restriction of the features to consider only pairs of adjacent tags. In a sense, we can think of this as a bigram language model, at the tag level. A natural question is how we might generalize Viterbi to tag trigrams:

$$\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, y_{m-2}, m). \tag{6.30}$$

One possibility is to take the Cartesian product of the tagset with itself, $\mathcal{Y}^{(2)} = \mathcal{Y} \times \mathcal{Y}$. The tags in this product space are ordered pairs, representing adjacent tags at the token level: for example, the tag $\langle \text{N}, \text{V} \rangle$ would represent a noun followed by a verb. Transitions between such tags must be consistent: we can have a transition from $\langle \text{N}, \text{V} \rangle$ to $\langle \text{V}, \text{N} \rangle$ (corresponding to the token-level tag sequence N V N), but not from $\langle \text{N}, \text{V} \rangle$ to $\langle \text{N}, \text{N} \rangle$, which would not correspond to any token-level tag sequence. This constraint can be enforced in the feature weights, with $\theta_{\langle\langle a,b\rangle,\langle c,d\rangle\rangle} = -\infty$ if $b \neq c$. The remaining feature weights can encode preferences for and against various tag trigrams.

The extension to trigrams changes the time and space complexity of the Viterbi algorithm. Recall that the time complexity of the Viterbi algorithm is $\mathcal{O}(MK^2)$: the size of the trellis is $M \times K$, and computing each element of the trellis requires maximizing over K possible predecessors. Now, the Cartesian product tag set is quadratic in the size of the original tag set, so the size of the trellis must be $M \times K^2$, with K representing the size of the original tagset. A naïve implementation of Viterbi in the Cartesian product tag space would compute each element of the trellis by maximizing over K^2 possible predecessors, for a total time complexity of $\mathcal{O}(MK^4)$. But this is more work than is necessary: even though the size of the tag-pair space is K^2, there are still only K possible predecessors for each position in the trellis, due to the consistency constraints mentioned in the previous paragraph. Careful design of the $\max$ operation in the Viterbi algorithm can exploit this, limiting the time complexity to $\mathcal{O}(MK^3)$. In general, the time and space complexity of the Viterbi algorithm grows exponentially with the order of the tag n-grams.

6.4 Hidden Markov Models

We now consider how to learn the weights $\boldsymbol{\theta}$ that parametrize the Viterbi sequence labeling algorithm. We begin with a probabilistic approach. Recall that the probabilistic Naïve Bayes classifier selects the label y to maximize $\text{p}(y \mid \boldsymbol{x}) \propto \text{p}(y, \boldsymbol{x})$. In probabilistic sequence labeling, our goal is similar: select the tag sequence that maximizes $\text{p}(\boldsymbol{y} \mid \boldsymbol{w}) \propto \text{p}(\boldsymbol{y}, \boldsymbol{w})$. Just as Naïve Bayes could be cast as a linear classifier maximizing $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y)$, we can cast our probabilistic classifier as a linear decision rule. Furthermore, the feature restriction in Equation 6.13 can be viewed as an Markov independence assumption on the elements of

$\boldsymbol{y}$. Thanks to this assumption, it is possible to perform inference using the Viterbi algorithm.

The Naïve Bayes algorithm was introduced as a generative model — a probabilistic story that explains the observed data as well as the hidden label. A similar story can be constructed for probabilistic sequence labeling: first, we draw the tags from a prior distribution, $\boldsymbol{y} \sim \text{p}(\boldsymbol{y})$; next, we draw the tokens from a conditional likelihood distribution, $\boldsymbol{w} \sim \text{p}(\boldsymbol{w} \mid \boldsymbol{y})$. However, for inference to be tractable, additional independence assumptions are required. Here we make two assumptions. First, the probability of each token depends only on its tag, and not on any other element in the sequence:

$$\text{p}(\boldsymbol{w} \mid \boldsymbol{y}) = \prod_{m}^{M} \text{p}(w_m \mid y_m). \tag{6.31}$$

Next, we introduce an independence assumption on the form of the prior distribution over labels: each label y_m depends only on its predecessor,

$$\text{p}(\boldsymbol{y}) = \prod_{m=1}^{M} \text{p}(y_m \mid y_{m-1}), \tag{6.32}$$

where $y_0 = \lozenge$ in all cases. Due to this **Markov** assumption, probabilistic sequence labeling models are known as **hidden Markov models** (HMMs). We now state the generative model under these independence assumptions,

- For $m \in (1, 2, \ldots, M)$,
  - draw $y_m \mid y_{m-1} \sim \text{Categorical}(\boldsymbol{\lambda}_{y_{m-1}})$;
  - draw $w_m \mid y_m \sim \text{Categorical}(\boldsymbol{\phi}_{y_m})$

This generative story formalizes the hidden Markov model. Given the parameters $\boldsymbol{\lambda}$ and $\boldsymbol{\phi}$, we can compute $\text{p}(\boldsymbol{w}, \boldsymbol{y})$ for any token sequence $\boldsymbol{w}$ and tag sequence $\boldsymbol{y}$. The HMM is often represented as a **graphical model** (Wainwright and Jordan, 2008), as shown in Figure 6.2. This representation makes the independence assumptions explicit: if a variable x is probabilistically conditioned on another variable y, then there is an arrow $x \rightarrow y$ in the diagram; otherwise, x and y are **conditionally independent** , given the other variables in the model.

It is important to reflect on the implications of the HMM independence assumptions. A non-adjacent pair of tags y_m and y_n are conditionally independent; if $m < n$ and we are given y_{n-1}, then y_m offers no additional information about y_m. However, if we are not given any information about the tags in a sequence, then all tags are probabilistically coupled.

$$\begin{array}{ccccccc} y_1 & \longrightarrow & y_2 & \longrightarrow & \cdots & \longrightarrow & y_M \\ \downarrow & & \downarrow & & & & \downarrow \\ w_1 & & w_2 & & \ldots & & w_M \end{array}$$

Figure 6.2: Graphical representation of the hidden Markov model. Arrows indicate probabilistic dependencies.

Estimation

The hidden Markov model has two groups of parameters:

Emission probabilities. The probability $\mathrm{p}_e(w_m \mid y_m; \phi)$ is the emission probability, since the words are treated as probabilistically "emitted", conditioned on the tags.

Transition probabilities. The probability $\mathrm{p}_t(y_m \mid y_{m-1}; \lambda)$ is the transition probability, since it assigns probability to each possible tag-to-tag transition.

Both of these groups of parameters are typically computed from relative frequency estimation on a labeled corpus,

$$\phi_{k,i} \triangleq \mathrm{p}(W_m = i \mid Y_m = k) = \frac{\text{count}(W_m = i, Y_m = k)}{\text{count}(Y_m = k)}$$
$$\lambda_{k,k'} \triangleq \mathrm{p}(Y_m = k' \mid Y_{m-1} = k) = \frac{\text{count}(Y_m = k', Y_{m-1} = k)}{\text{count}(Y_{m-1} = k)}.$$

Smoothing is more important for the emission probability than the transition probability, because the event space is much larger. Smoothing techniques such as additive smoothing, interpolation, and backoff (see chapter 5) can all be applied here.

Inference

The goal of inference in the hidden Markov model is to find the highest probability tag sequence,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\text{argmax}}\, \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}). \tag{6.33}$$

As in Naïve Bayes, it is equivalent to find the tag sequence with the highest **log**-probability, since the log function is monotonically increasing. It is furthermore equivalent to maximize the joint probability $\mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})\mathrm{p}(\boldsymbol{w}) \propto \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$, which is proportional to the conditional probability. Therefore, we can reformulate the inference problem as,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{\text{argmax}} \log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}). \tag{6.34}$$

We can now apply the HMM independence assumptions:

$$\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \log \mathrm{p}(\boldsymbol{y}) + \log \mathrm{p}(\boldsymbol{w} \mid \boldsymbol{y}) \tag{6.35}$$

$$= \sum_{m=1}^{M} \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) \tag{6.36}$$

$$= \sum_{m=1}^{M} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m}. \tag{6.37}$$

This log probability can be rewritten as a dot product of weights and features,

$$\log \mathrm{p}(\boldsymbol{y}, \boldsymbol{w}) = \sum_{m=1}^{M} \log \lambda_{y_m, y_{m-1}} + \log \phi_{y_m, w_m} \tag{6.38}$$

$$= \sum_{m=1}^{M} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m), \tag{6.39}$$

where the feature function $\boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \{\langle y_m, y_{m-1}\rangle, \langle y_m, w_m\rangle\}$, and the weight vector θ encodes the log-parameters $\log \boldsymbol{\lambda}$ and $\log \boldsymbol{\phi}$.

This derivation shows that HMM inference can be viewed as an application of the Viterbi decoding algorithm, given an appropriately defined feature function and weight vector. The local product $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$ can be interpreted probabilistically,

$$\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \log \mathrm{p}_y(y_m \mid y_{m-1}) + \log \mathrm{p}_{w|y}(w_m \mid y_m) \tag{6.40}$$

$$= \log \mathrm{p}(y_m, w_m \mid y_{m-1}) \tag{6.41}$$

Now recall the definition of the Viterbi variables,

$$v_m(y) = \max_{\boldsymbol{y}_{1:m-1}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y, y_{m-1}, m) + \sum_{n=1}^{m-1} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_n, y_{n-1}, n) \tag{6.42}$$

$$v_m(y) = \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(y_m, w_m \mid y_{m-1}) + \sum_{n=1}^{m-1} \log \mathrm{p}(y_n, w_n \mid y_{n-1}) \tag{6.43}$$

$$= \max_{\boldsymbol{y}_{1:m-1}} \log \mathrm{p}(\boldsymbol{y}_{1:m-1}, Y_m = y, \boldsymbol{w}_{1:m}). \tag{6.44}$$

In words, the Viterbi variable $v_m(y)$ is the log probability of the best tag sequence ending in $Y_m = y$, joint with the word sequence $\boldsymbol{w}_{1:m}$. The log probability of the best complete tag sequence is therefore,

$$\max_{\boldsymbol{y}_{1:M}} \log \mathrm{p}(\boldsymbol{y}_{1:M}, \boldsymbol{w}_{1:M}) = \max_{y_M} \log \mathrm{p}_y(\blacklozenge \mid y_M) + v_M(y_M). \tag{6.45}$$

The Viterbi algorithm can be implemented using probabilities, rather than log probabilities. In this case, each $v_m(y)$ is equal to,

$$
\begin{aligned}
v_m(y) &= \max_{\boldsymbol{y}_{1:m-1}} \mathrm{p}(\boldsymbol{y}_{1:m-1}, Y_m = y, \boldsymbol{w}_{1:m}) && (6.46)\\
&= \max_{y_{m-1}} \mathrm{p}(y_m, w_m \mid y_{m-1}) \max_{\boldsymbol{y}_{1:m-2}} \mathrm{p}(y_{1:m-2}, y_{m-1}, \boldsymbol{w}_{1:m-1}) && (6.47)\\
&= \max_{y_{m-1}} \mathrm{p}(y_m, w_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && (6.48)\\
&= \mathrm{p}(w_m \mid y_m) \times \max_{y_{m-1}} \mathrm{p}(y_m \mid y_{m-1}) \times v_{m-1}(y_{m-1}) && (6.49)\\
&= \max_{y_{m-1}} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)) \times v_{m-1}(y_{m-1}). && (6.50)
\end{aligned}
$$

In the final line, we use the fact that $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m) = \log \mathrm{p}(y_m, w_m \mid y_{m-1})$, and exponentiate the dot product to obtain the probability.

In practice, the probabilities tend towards zero over long sequences, so the log-probability version of Viterbi is more practical from the standpoint of numerical stability. However, this version of the algorithm is often taught first, since it can be explained directly in terms of probabilities. It also connects to a broader literature on inference in graphical models. Each Viterbi variable is computed by **maximizing** over a set of **products**; thus, the Viterbi algorithm is a special case of the **max-product algorithm** for inference in graphical models (Wainwright and Jordan, 2008).

The Forward Algorithm

In an influential survey, Rabiner (1989) defines three problems for hidden Markov models:

Decoding Find the best tags $\boldsymbol{y}$ for a sequence $\boldsymbol{w}$.

Likelihood Compute the marginal probability $\mathrm{p}(\boldsymbol{w}) = \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y})$.

Learning Given only unlabeled data $\{\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_D\}$, estimate the transition and emission distributions.

The Viterbi algorithm solves the decoding problem. We'll talk about the learning problem in section 6.6. Let's now consider how to compute the marginal likelihood $\mathrm{p}(\boldsymbol{w}) = \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y})$, which involves summing over all possible tag sequences. There are at least two reasons we might want to do this:

Language modeling Note that the probability $\mathrm{p}(\boldsymbol{w})$ is also computed by the language models that were discussed in chapter 5. In those language models, we used only unlabeled corpora, conditioning each token w_m on previous tokens. An HMM-based language model would leverage a corpus of part-of-speech annotations, and therefore might be expected to generalize better than an n-gram language model —

for example, it would be more likely to assign positive probability to a nonsense grammatical sentence like *colorless green ideas sleep furiously.*

Tag marginals It is often important to compute marginal probabilities of individual tags, $\text{p}(y_m \mid \boldsymbol{w}_{1:M})$. This is the probability distribution over tags for token m, conditioned on all of the words $\boldsymbol{w}_{1:M}$. For example, we might like the know the probability that a given word is tagged as a verb, regardless of how all the other words are tagged. We will discuss how to compute this probability in section 6.5, but as a preview, we will use the following form,

$$\text{p}(y_m \mid \boldsymbol{w}_{1:M}) = \frac{\text{p}(y_m, \boldsymbol{w}_{1:M})}{\text{p}(\boldsymbol{w}_{1:M})}, \tag{6.51}$$

which involves the marginal likelihood in the denominator.

We can compute the marginal likelihood using a dynamic program that is nearly identical to the Viterbi algorithm. We will use probabilities for now, and show the conversion to log-probabilities later. The core of the algorithm is to compute a set of **forward variables** ,

$$\alpha_m(y) \triangleq \text{p}(Y_m = y, \boldsymbol{w}_{1:m}). \tag{6.52}$$

From this definition, we can compute the marginal likelihood by summing over the final forward variables,

$$\begin{aligned}
\text{p}(\boldsymbol{w}) &= \sum_y \text{p}(Y_M = y, \boldsymbol{w}_{1:M}) && (6.53)\\
&= \sum_y \alpha_M(y). && (6.54)
\end{aligned}$$

To capture the probability of terminating the sequence on each possible tag Y_M, we can pad the end of $\boldsymbol{w}$ with an extra token ■, which can only be emitted from the stop tag ♦.

The forward variables themselves can be computed recursively,

$$\begin{aligned}
\alpha_m(k) &= \Pr(Y_m = k, \boldsymbol{W}_{1:m} = \boldsymbol{w}_{1:m}) && (6.55)\\
&= \text{p}(w_m \mid Y_m = k) \times \Pr(Y_m = k \mid \boldsymbol{w}_{1:m-1}) && (6.56)\\
&= \text{p}(w_m \mid Y_m = k) \times \sum_{k'} \Pr(Y_m = k, Y_{m-1} = k' \mid \boldsymbol{w}_{1:m-1}) && (6.57)\\
&= \text{p}(w_m \mid Y_m = k) \times \sum_{k'} \Pr(Y_m = k \mid Y_{m-1} = k') \times \Pr(Y_{m-1} = k' \mid \boldsymbol{w}_{1:m-1}) && (6.58)\\
&= \text{p}(w_m \mid Y_m = k) \times \sum_{k'} \Pr(Y_m = k \mid Y_{m-1} = k') \times \alpha_{m-1}(k') && (6.59)\\
&= \sum_{k'} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}_{1:M}, Y_m = k, Y_{m-1} = k', m)) \times \alpha_{m-1}(k'). && (6.60)
\end{aligned}$$

The derivation relies on the independence assumptions in the hidden Markov model: W_m depends only on Y_m, and Y_m is conditionally independent from $W_{1:m-1}$ and all tags, given Y_{m-1}. We complete the derivation by introducing Y_{m-1} and summing over all possible values, and by then applying the chain rule to obtain the final recursive form.

Procedurally, we compute the forward variables in just the same way as we compute the Viterbi variables: we first compute all $\alpha_1(\cdot)$, then all $\alpha_2(\cdot)$, and so on. We initialize each $\alpha_0(k) = \text{p}(Y_m = k \mid Y_{m-1} = \lozenge)$, to capture the transition probability from the start symbol. Comparing Equation 6.59 to Equation 6.49, the sole difference is that instead of maximizing over possible values of Y_{m-1}, we sum. Just as the Viterbi algorithm is a special case of the max-product algorithm for inference in graphical models, the forward algorithm is a special case of the **sum-product** algorithm for computing marginal likelihoods.

In practice, it is numerically more stable to compute the marginal log-probability. In the log domain, the forward recurrence is,

$$\begin{aligned}
\alpha_m(k) &\triangleq \log \text{p}(\boldsymbol{w}_{1:m}, Y_m = k) && (6.61)\\
&= \log \sum_{k'} \exp(\log \text{p}(w_m \mid Y_m = k) + \log \Pr(Y_m = k \mid Y_{m-1} = k') \\
&\quad + \log \text{p}(\boldsymbol{w}_{1:m-1}, Y_{m-1} = k')) && (6.62)\\
&= \log \sum_{k'} \exp(\log \text{p}(w_m \mid Y_m = k) + \log \Pr(Y_m = k \mid Y_{m-1} = k') \\
&\quad + \alpha_{m-1}(k')) && (6.63)\\
&= \log \sum_{k'} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}_{1:M}, Y_m = k, Y_{m-1} = k', m) + \alpha_{m-1}(k')). && (6.64)
\end{aligned}$$

Scientific programming libraries provide numerically robust implementations of the log-sum-exp function, which should prevent overflow and underflow from exponentiation.

Semiring Notation and the Generalized Viterbi Algorithm

We have now seen the Viterbi and Forward recurrences, each of which can be performed over probabilities or log probabilities. These four recurrences are closely related, and can in fact be expressed as a single recurrence in a more general notation, known as **semiring algebra** . We use the symbol $\oplus$ to represent generalized addition, and the symbol $\otimes$ to represent generalized multiplication.[2] Given these operators, we can denote a generalized Viterbi recurrence as,

$$v_m(k) = \bigoplus_{k'} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, Y_m = k, Y_{m-1} = k', m) \otimes v_{m-1}(k'). \quad (6.65)$$

[2] In a semiring, the addition and multiplication operators must both obey associativity, and multiplication must distribute across addition; the addition operator must be commutative; there must be additive and multiplicative identities $\overline{0}$ and $\overline{1}$; and there must be a multiplicative annihilator $\overline{0}$, such that $a \otimes \overline{0} = \overline{0}$.

Each recurrence that we have seen so far is a special case of this generalized Viterbi recurrence:

- In the max-product Viterbi recurrence over probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to multiplication.
- In the Forward recurrence over probabilities, the $\oplus$ operation corresponds to addition, and the $\otimes$ operation corresponds to multiplication.
- In the max-product Viterbi recurrence over log-probabilities, the $\oplus$ operation corresponds to maximization, and the $\otimes$ operation corresponds to addition. (This is sometimes called the **tropical semiring** , in honor of the Brazilian mathematician Imre Simon.)
- In the Forward recurrence over log-probabilities, the $\oplus$ operation corresponds to log-addition, $a \oplus b = \log(e^a + e^b)$. The $\otimes$ operation corresponds to addition.

The mathematical abstraction offered by semiring notation can be applied to the software implementations of these algorithms, yielding concise and modular implementations. The OpenFST library (Allauzen et al., 2007) is an example of a software package in which the algorithms are parametrized by the choice of semiring.

6.5 Discriminative sequence labeling

Today, hidden Markov models are rarely used for supervised sequence labeling. This is because HMMs are limited to only two phenomena:

- Word-tag probabilities, via the emission probability $\text{p}_E(w_m \mid y_n)$;
- local context, via the transition probability $\text{p}_T(y_m \mid y_{m-1})$.

However, as we have seen, the Viterbi algorithm can be applied to much more general feature sets, as long as the decomposition $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$ is observed. In this section, we discuss methods for learning the weights on such features. However, let's first pause to ask what additional features might be needed.

Word affix features. Consider the problem of part-of-speech tagging on the first four lines of the poem *Jabberwocky* (Carroll, 1917):

(6.3) 'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

Many of these words are made up, so you would have no information about their probabilities of being associated with any particular part of speech. Yet it is not so hard to see what their grammatical roles might be in this passage. Context helps: for example, the word *slithy* follows the determiner *the*, and therefore is likely to be a noun or adjective. Which do you think is more likely? The suffix *-thy* is found in a number of adjectives — e.g., *frothy,healthy,pithy,worthy*. The suffix is also found in a handful of nouns — e.g., *apathy,sympathy* — but nearly all of these nouns contain *-pathy*, unlike *slithy*. The suffix gives some evidence that *slithy* is an adjective, and indeed it is: later in the text we find that it is a combination of the adjectives *lithe* and *slimy*.[3]

Fine-grained context. Another useful source of information is fine-grained context — that is, contextual information that is more specific than the previous tag. For example, consider the noun phrases *this fish* and *these fish*. Many part-of-speech tagsets distinguish between singular and plural nouns, but do not distinguish between singular and plural determiners; for example, the Penn Treebank tagset follows these conventions. A hidden Markov model would be unable to correctly label *fish* as singular or plural in both of these cases, because it only has access to two features: the preceding tag (determiner in both cases) and the word (*fish* in both cases). The classification-based tagger discussed in section 6.1 had the ability to use preceding and succeeding words as features, and we would like to incorporate this information into a sequence labeling algorithm.

Example Suppose we have the tagging D J N (determiner, adjective, noun) for the sequence *the slithy toves* in Jabberwocky, so that

$$\begin{aligned}\boldsymbol{w} &= \textit{the slithy toves}\\ \boldsymbol{y} &= \text{D J N}.\end{aligned}$$

We now create the feature vector for this example, assuming that we have word-tag features (indicated by prefix W), tag-tag features (indicated by prefix T), and suffix features (indicated by prefix M). We assume access to a method for extracting the suffix *-thy* from *slithy*, *-es* from *toves*, and $\varnothing$ from *the*, indicating that this word has no suffix. The resulting feature vector is,

$$\begin{aligned}\boldsymbol{f}(\text{the slithy toves}, \text{D J N}) = \{&\langle W : \text{the}, \text{D}\rangle, \langle M : \varnothing, \text{D}\rangle, \langle T : \lozenge, \text{D}\rangle\\ &\langle W : \text{slithy}, \text{J}\rangle, \langle M : \text{-thy}, \text{J}\rangle, \langle T : \text{D}, \text{J}\rangle\\ &\langle W : \text{toves}, \text{N}\rangle, \langle M : \text{-es}, \text{N}\rangle, \langle T : \text{J}, \text{N}\rangle\\ &\langle T : \text{N}, \blacklozenge\rangle\}.\end{aligned}$$

[3] **Morphology** is the study of how words are formed from smaller linguistic units. Computational approaches to morphological analysis are touched on in chapter 8; Bender (2013) provides a good overview of the underlying linguistic principles.

We now consider several discriminative methods for learning feature weights in sequence labeling. In chapter 2, we considered three types of discriminative classifiers: perceptron, support vector machine, and logistic regression. Each of these classifiers has a structured equivalent, enabling it to be trained from labeled sequences rather than individual tokens.

Structured perceptron

The perceptron classifier updates its weights by increasing the weights for features that are associated with the correct label, and decreasing the weights for features that are associated with incorrectly predicted labels:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y) \tag{6.66}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{x}, y) - \boldsymbol{f}(\boldsymbol{x}, \hat{y}). \tag{6.67}$$

We can apply exactly the same update in the case of structure prediction,

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})}{\text{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.68}$$

$$\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} + \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) - \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}}). \tag{6.69}$$

This learning algorithm is called **structured perceptron** , because it learns to predict the structured output $\boldsymbol{y}$. The key difference is that instead of computing $\hat{\boldsymbol{y}}$ by enumerating the entire set $\mathcal{Y}$, we use the Viterbi algorithm to search this set efficiently. In this case, the output structure is the sequence of tags $(y_1, y_2, \ldots, y_M)$; the algorithm can be applied to other structured outputs as long as efficient inference is possible. As in perceptron classification, weight averaging is crucial to get good performance (see section 2.1).

Example For the example *They can fish*, suppose the reference tag sequence is N V V, but our tagger incorrectly returns the tag sequence N V N. Given **feature templates** $\langle w_m, y_m \rangle$ and $\langle y_{m-1}, y_m \rangle$, the corresponding structured perceptron update is:

$$\theta_{\langle \textit{fish}, \text{V} \rangle} \leftarrow \theta_{\langle \textit{fish}, \text{V} \rangle} + 1 \tag{6.70}$$

$$\theta_{\langle \textit{fish}, \text{N} \rangle} \leftarrow \theta_{\langle \textit{fish}, \text{N} \rangle} - 1 \tag{6.71}$$

$$\theta_{\langle \text{V}, \text{V} \rangle} \leftarrow \theta_{\langle \text{V}, \text{V} \rangle} + 1 \tag{6.72}$$

$$\theta_{\langle \text{V}, \text{N} \rangle} \leftarrow \theta_{\langle \text{V}, \text{N} \rangle} - 1 \tag{6.73}$$

$$\theta_{\langle \text{V}, \blacklozenge \rangle} \leftarrow \theta_{\langle \text{V}, \blacklozenge \rangle} + 1 \tag{6.74}$$

$$\theta_{\langle \text{N}, \blacklozenge \rangle} \leftarrow \theta_{\langle \text{N}, \blacklozenge \rangle} - 1. \tag{6.75}$$

Structured Support Vector Machines

Large-margin classifiers such as the support vector machine improve on the perceptron by learning weights that push the classification boundary away from the training instances. In many cases, large-margin classifiers outperform the perceptron, so we would like to apply similar ideas to sequence labeling. A support vector machine in which the output is a structured object, such as a sequence, is called a **structured support vector machine** (Tsochantaridis et al., 2004).[4]

In classification, we formalized the large-margin constraint as,

$$\forall y \neq y^{(i)}, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y^{(i)}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{x}, y) \geq 1, \tag{6.76}$$

which says that we require a margin of at least 1 between the scores for all labels y that are not equal to the correct label $y^{(i)}$. The weights $\boldsymbol{\theta}$ are then learned by constrained optimization (see section 2.2); for example, the PEGASOS algorithm (Shalev-Shwartz et al., 2007) applied stochastic subgradient descent to a Lagrangian expression that combines the margin constraints with a regularizer.

We can apply this idea to sequence labeling by formulating an equivalent set of constraints for all possible labelings $\mathcal{Y}(\boldsymbol{w})$ for an input $\boldsymbol{w}$. However, there are two problems with this idea. First, in sequence labeling, some predictions are more wrong than others: we may miss only one tag out of fifty, or we may get all fifty wrong. We would like our learning algorithm to be sensitive to this difference. Second, the number of contraints is equal to the number of possible labelings, which is exponentially large in the length of the sequence.

The first problem can be addressed by adjusting the constraint to require larger margins for more serious errors. Let $c(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}) \geq 0$ represent the **cost** of predicting label $\hat{\boldsymbol{y}}$ when the true label is $\boldsymbol{y}^{(i)}$. We can then generalize the margin constraint,

$$\forall \boldsymbol{y} \neq \boldsymbol{y}^{(i)}, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \geq c(\boldsymbol{y}^{(i)}, \boldsymbol{y}). \tag{6.77}$$

This cost-augmented margin constraint specializes to the constraint in Equation 6.76 if we choose the delta function $c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \delta(\boldsymbol{y}^{(i)} \neq \boldsymbol{y})$. For sequence labeling, we can instead use a structured cost function, such as the **Hamming cost** ,

$$c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) = \sum_{m=1}^{M} \delta(y_m^{(i)} \neq y_m). \tag{6.78}$$

With this cost function, we require that the true labeling be seperated from the alternatives by a margin that is proportional to the number of incorrect tags in each alternative labeling. Other cost functions are possible as well.

[4] This model is also known as a **max-margin Markov network** (Taskar et al., 2003), emphasizing that the scoring function is constructed from a sum of components, which are Markov independent.

The second problem is that the number of constraints is exponential in the length of the sequence. This can be addressed by focusing on the prediction $\hat{\boldsymbol{y}}$ that *maximally* violates the margin constraint. We find this prediction by solving the following **cost-augmented decoding** problem:

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}}{\text{argmax}}\, \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) - \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \tag{6.79}$$

$$= \max_{\boldsymbol{y} \neq \boldsymbol{y}^{(i)}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}), \tag{6.80}$$

where in the second line we drop the term $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$, which is constant in $\boldsymbol{y}$.

We can now formulate the margin constraint for sequence labeling,

$$\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \Big(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}) + c(\boldsymbol{y}^{(i)}, \boldsymbol{y}) \Big) \geq 0. \tag{6.81}$$

If the score for $\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ is greater than the cost-augmented score for all alternatives, then the constraint will be met. Therefore we can maximize over the entire set $\mathcal{Y}(\boldsymbol{w})$, meaning that we can apply Viterbi directly.[5]

The name "cost-augmented decoding" is due to the fact that the objective includes the standard decoding problem, $\max_{\hat{\boldsymbol{y}}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \hat{\boldsymbol{y}})$, plus an additional term for the cost. Essentially, we want to train against predictions that are strong and wrong: they should score highly according to the model, yet incur a large loss with respect to the ground truth. We can then adjust the weights to reduce the score of these predictions.

For cost-augmented decoding to be tractable, the cost function must decompose into local parts, just as the feature function $\boldsymbol{f}(\cdot)$ does. The Hamming cost, defined above, obeys this property. To solve this cost-augmented decoding problem using the Hamming cost, we can simply add features $f_m(y_m) = \delta(y_m \neq y_m^{(i)})$, and assign a weight of 1 to these features. Decoding can then be performed using the Viterbi algorithm.

Are there cost functions that do not decompose into local parts? Suppose we want to assign a constant loss to any prediction $\hat{\boldsymbol{y}}$ in which k or more predicted tags are incorrect, and zero loss otherwise. This loss function is combinatorial over the predictions, and thus we cannot decompose it into parts.

As with support vector machine classifiers, we can formulate the learning problem as an unconstrained *primal form*, which combines a regularization term on the weights and a Lagrangian for the constraints:

$$\min_{\boldsymbol{\theta}} \quad \frac{1}{2}||\boldsymbol{\theta}||_2^2 - C \left(\sum_i \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) - \max_{\hat{\boldsymbol{y}} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \Big[\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \hat{\boldsymbol{y}}) + c(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}) \Big] \right), \tag{6.82}$$

[5] To maximize over the set $\mathcal{Y}(\boldsymbol{w}) \setminus \boldsymbol{y}^{(i)}$ we would need to an alternative version of Viterbi that returns the k-best predictions. K-best Viterbi may be useful for other reasons — for example, in interactive applications, it can be helpful to show the user multiple possible taggings. The design of k-best Viterbi is left an exercise.

In this formulation, C is a parameter that controls the tradeoff between the regularization term and the margin constraints. A number of optimization algorithms have been proposed for structured support vector machines, some of which are discussed in section 2.2. An empirical comparison by Kummerfeld et al. (2015) shows that stochastic subgradient descent — which is relatively easy to implement — is highly competitive, especially on the sequence labeling task of named entity recognition.

Conditional random fields

Structured perceptron is easy to implement, and structured support vector machines give excellent performance. However, sometimes we need to compute probabilities over labelings, $\text{p}(\boldsymbol{y} \mid \boldsymbol{w})$, and we would like to do this in a discriminative way. The **Conditional Random Field** (CRF; Lafferty et al., 2001) is a conditional probabilistic model for sequence labeling; just as structured perceptron is built on the perceptron classifier, conditional random fields are built on the logistic regression classifier.[6] The basic probability model is,

$$\text{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}))}{\sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}'))}. \tag{6.83}$$

This is almost identical to logistic regression, but because the label space is now tag sequences, we require efficient algorithms for both **decoding** (searching for the best tag sequence given a sequence of words $\boldsymbol{w}$ and a model $\boldsymbol{\theta}$) and for **normalizing** (summing over all tag sequences). These algorithms will be based on the usual locality assumption on the feature function, $\boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y}) = \sum_{m=1}^{M} \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$.

Decoding in CRFs

Decoding — finding the tag sequence $\hat{\boldsymbol{y}}$ that maximizes $\text{p}(\boldsymbol{y} \mid \boldsymbol{w})$ — is a direct application of the Viterbi algorithm. The key observation is that the decoding problem does not

[6] The name "Conditional Random Field" is derived from **Markov random fields**, a general class of models in which the probability of a configuration of variables is proportional to a product of scores across pairs (or more generally, cliques) of variables in a **factor graph**. In sequence labeling, the pairs of variables include all adjacent tags $\langle y_m, y_{m-1} \rangle$. The probability is **conditioned** on the words $\boldsymbol{w}_{1:M}$, which are always observed, motivating the term "conditional" in the name.

depend on the denominator of $\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w})$,

$$\begin{aligned}
\hat{\boldsymbol{y}} &= \operatorname*{argmax}_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \\
&= \operatorname*{argmax}_{\boldsymbol{y}} \log \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) \\
&= \operatorname*{argmax}_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}) - \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w})} e^{\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}', \boldsymbol{w})} \\
&= \operatorname*{argmax}_{\boldsymbol{y}} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{w}).
\end{aligned}$$

This is identical to the decoding problem for structured perceptron, so the same Viterbi recurrence as defined in Equation 6.26 can be used.

Learning in CRFs

As with logistic regression, we learn the weights $\boldsymbol{\theta}$ by minimizing the regularized negative log conditional probability,

$$\ell = \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \log \mathrm{p}(\boldsymbol{y}^{(i)} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) \tag{6.84}$$

$$= \frac{\lambda}{2}||\boldsymbol{\theta}||^2 - \sum_{i=1}^{N} \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}) + \log \sum_{\boldsymbol{y}' \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}^{(i)}, \boldsymbol{y}')\right), \tag{6.85}$$

where λ controls the amount of regularization. We will optimize $\boldsymbol{\theta}$ by moving along the gradient of this loss. Probabilistic programming environments, such as THEANO (Bergstra et al., 2010) and TORCH (Collobert et al., 2011), can compute this gradient using automatic differentiation. However, it is worth deriving the gradient to understand how this model works, and why learning is computationally tractable.

As in logistic regression, the gradient includes a difference between observed and expected feature counts:

$$\frac{d\ell}{d\theta_j} = \lambda\theta_j + \sum_{i=1}^{N} E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] - f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)}), \tag{6.86}$$

where $f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}^{(i)})$ refers to the count of feature j for token sequence $\boldsymbol{w}^{(i)}$ and tag sequence $\boldsymbol{y}^{(i)}$.

The expected feature counts are computed by summing over all possible labelings of the word sequence,

$$E[f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y})] = \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w}^{(i)})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}^{(i)}; \boldsymbol{\theta}) f_j(\boldsymbol{w}^{(i)}, \boldsymbol{y}) \tag{6.87}$$

This looks bad: it is a sum over an exponential number of labelings. To solve this problem, we again rely on the assumption that the overall feature vector decomposes into a sum of local feature vectors, which we exploit to compute the expected feature counts as a sum across the sequence:

$$E[f_j(\boldsymbol{w}, \boldsymbol{y})] = \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, \boldsymbol{y}) \tag{6.88}$$

$$= \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) \sum_{m=1}^{M} f_j(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.89}$$

$$= \sum_{m=1}^{M} \sum_{\boldsymbol{y} \in \mathcal{Y}(\boldsymbol{w})} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, y_m, y_{m-1}, m) \tag{6.90}$$

$$= \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} \sum_{\boldsymbol{y}: y_{m-1}=k', y_m=k} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) f_j(\boldsymbol{w}, k, k', m) \tag{6.91}$$

$$= \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} f_j(\boldsymbol{w}, k, k', m) \sum_{\boldsymbol{y}: y_{m-1}=k', y_m=k} \mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}; \boldsymbol{\theta}) \tag{6.92}$$

$$= \sum_{m=1}^{M} \sum_{k,k'}^{\mathcal{Y}} f_j(\boldsymbol{w}, k', k, m) \Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta}). \tag{6.93}$$

This derivation works by interchanging the sum over sequences with the sum over m. At each position in the sequence, we need only the marginal probability of the tag bigram, $\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta})$. These tag bigram marginals are also used in unsupervised approaches to sequence labeling. In principle, these marginals still require a sum over the exponentially many label sequences in which $Y_{m-1} = k'$ and $Y_m = k$. However, they can be computed efficiently using the **forward-backward algorithm** .

***Forward-backward algorithm**

Recall that we previously used the Forward algorithm to compute marginal probabilities $\mathrm{p}(y_m, \boldsymbol{w}_{1:m})$ in the hidden Markov model. We now derive a more general version of the algorithm, in which label sequences are scored in terms of **potentials** $\psi_m(k, k')$:

$$\psi_m(k, k') \triangleq \exp(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, k, k', m)) \tag{6.94}$$

$$\mathrm{p}(\boldsymbol{y} \mid \boldsymbol{w}) = \frac{\prod_{m=1}^{M} \psi_m(y_m, y_{m-1})}{\sum_{\boldsymbol{y}'} \prod_{m=1}^{M} \psi_m(y'_m, y'_{m-1})}. \tag{6.95}$$

It should be clear that Equation 6.95 simply expresses the CRF conditional likelihood, under a change of notation.

Using this notation, we can compute the tag bigram marginals as,

$$\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}; \boldsymbol{\theta}) = \frac{\sum_{\boldsymbol{y}: y_m = k, y_{m-1} = k'} \prod_{n=1}^{M} \psi_n(y_n, y_{n-1})}{\sum_{\boldsymbol{y}'} \prod_{n=1}^{M} \psi_n(y'_n, y'_{n-1})}. \tag{6.96}$$

where the denominator is the marginal $\mathrm{p}(\boldsymbol{w}_{1:M}) = \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y}_{1:M})$, sometimes known as the **partition function** .[7] Let us now consider how to compute each of these terms efficiently.

Computing the numerator In Equation 6.96, we sum over all tag sequences that include the transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$. Because we are only interested in sequences that include this arc, we can decompose this sum into three parts: the sum over **prefixes** $\boldsymbol{y}_{1:m-1}$, the transition $(Y_{m-1} = k') \rightarrow (Y_m = k)$, and the sum over **suffixes** $\boldsymbol{y}_{m:M}$,

$$\begin{aligned}\sum_{\boldsymbol{y}: Y_m = k, Y_{m-1} = k'} \prod_{n=1}^{M} \psi_n(y_n, y_{n-1}) = & \sum_{\boldsymbol{y}_{1:m-1}: y_{m-1} = k'} \prod_{n=1}^{m-1} \psi_n(y_n, y_n - 1) \\ & \times \psi_m(k, k') \\ & \times \sum_{\boldsymbol{y}_{m:M}: y_m = k} \prod_{n=m+1}^{M} \psi_n(y_n, y_{n-1}).\end{aligned} \tag{6.97}$$

The result is product of three terms: a score for getting to the position $(Y_{m-1} = k')$, a score for the transition from k' to k, and a score for finishing the sequence from $(Y_m = k)$. Let us define the first term as a **forward variable**,

$$\alpha_m(k) \triangleq \sum_{\boldsymbol{y}_{1:m}: y_m = k} \prod_{n=1}^{m} \psi_n(y_n, y_{n-1}) \tag{6.98}$$

$$= \sum_{k'} \psi_m(k, k') \sum_{\boldsymbol{y}_{1:m-1}: y_{m-1} = k'} \prod_{n=1}^{m-1} \psi_n(y_n, y_{n-1}) \tag{6.99}$$

$$= \sum_{k'} \psi_m(k, k') \alpha_{m-1}(k'). \tag{6.100}$$

Thus, we compute the forward variables while moving from left to right over the trellis. This forward recurrence is a generalization of the forward recurrence defined in section 6.4. If we set $\psi_m(k, k') = \mathrm{p}_E(w_m \mid Y_m = k) \Pr(Y_m = k \mid Y_{m-1} = k')$, we exactly recover the HMM forward variable $\alpha_m(k) = \mathrm{p}(\boldsymbol{w}_{1:m}, Y_m = k)$.

[7] The terminology of "potentials" and "partition functions" is due to a connection with statistical mechanics (Bishop, 2006).

The third term of Equation 6.97 can also be defined recursively, this time moving over the trellis from right to left. The resulting recurrence is called the **backward algorithm**:

$$\beta_{m-1}(k) \triangleq \sum_{\boldsymbol{y}_{m-1:M}:y_{m-1}=k} \prod_{n=m}^{M} \psi_n(y_n, y_{n-1}) \tag{6.101}$$

$$= \sum_{k'} \psi_m(k', k) \sum_{\boldsymbol{y}_{m:M}:y_m=k'} \prod_{n=m+1}^{M} \psi_n(y_n, y_{n-1}) \tag{6.102}$$

$$= \sum_{k'} \psi_m(k', k)\beta_m(k'). \tag{6.103}$$

In practice, numerical stability requires that we use log-potentials rather than potentials, $\log \psi_m(y_m, y_{m-1}) = \boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)$. Then the sums must be replaced with log-sum-exp:

$$\log \alpha_m(k) = \log \sum_{k'} \exp\left(\log \psi_m(k, k') + \log \alpha_{m-1}(k')\right) \tag{6.104}$$

$$\log \beta_{m-1}(k) = \log \sum_{k'} \exp\left(\log \psi_m(k', k) + \log \beta_m(k')\right)). \tag{6.105}$$

Both the forward and backward algorithm operate on the trellis, which implies a space complexity $\mathcal{O}(MK)$. Because they require computing a sum over K terms at each node in the trellis, their time complexity is $\mathcal{O}(MK^2)$.

Computing the normalization term The normalization term, sometimes abbreviated as Z, can be written as,

$$Z \triangleq \sum_{\boldsymbol{y}} \mathrm{p}(\boldsymbol{w}, \boldsymbol{y}) \tag{6.106}$$

$$= \sum_{\boldsymbol{y}} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, \boldsymbol{y})\right) \tag{6.107}$$

$$= \sum_{\boldsymbol{y}} \prod_{m=1}^{M} \exp\left(\boldsymbol{\theta}^\top \boldsymbol{f}(\boldsymbol{w}, y_m, y_{m-1}, m)\right) \tag{6.108}$$

$$= \sum_{\boldsymbol{y}} \prod_{m=1}^{M} \psi_m(y_m, y_{m-1}). \tag{6.109}$$

This term can be computed directly from either the forward or backward probabilities:

$$Z = \sum_{\boldsymbol{y}} \prod_{m=1}^{M} \psi_m(y_m, y_{m-1}) \tag{6.110}$$

$$= \alpha_{M+1}(\blacklozenge) \tag{6.111}$$

$$= \beta_0(\lozenge). \tag{6.112}$$

CRF learning: wrapup Having computed the forward and backward variables, we can compute the desired marginal probability as,

$$P(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M}) = \frac{\alpha_{m-1}(k')\psi_m(k, k')\beta_m(k)}{Z}. \tag{6.113}$$

This computation is known as the **forward-backward algorithm.** From the resulting marginals, we can compute the feature expectations $E[f_j(\boldsymbol{w}, \boldsymbol{y})]$; from these expectations, we compute a gradient on the weights $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. Stochastic gradient descent or quasi-Newton optimization can then be applied. As the optimization algorithm changes the weights, the potentials change, and therefore so do the marginals. Each iteration of the optimization algorithm therefore requires recomputing the forward and backward variables for each training instance. [8]

6.6 *Unsupervised sequence labeling

In unsupervised sequence labeling, we want to induce a Hidden Markov Model from a corpus of unannotated text $\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \ldots, \boldsymbol{w}^{(N)}$. This is an example of the general problem of **structure induction**, which is the unsupervised version of **structure prediction**. The tags that result from unsupervised sequence labeling might be useful for some downstream task, or they might help us to better understand the language's inherent structure.

Unsupervised learning in hidden Markov models can be performed using the **Baum-Welch algorithm**, which combines forward-backward with expectation-maximization (EM). In the M-step, we compute the HMM parameters from expected counts:

$$\Pr(W = i \mid Y = k) = \phi_{k,i} = \frac{E[\text{count}(W = i, Y = k)]}{E[\text{count}(Y = k)]}$$

$$\Pr(Y_m = k \mid Y_{m-1} = k') = \lambda_{k',k} = \frac{E[\text{count}(Y_m = k, Y_{m-1} = k')]}{E[\text{count}(Y_{m-1} = k')]}$$

[8] The `CRFsuite` package implements several learning algorithms for CRFs (`http://www.chokkan.org/software/crfsuite/`).

The expected counts are computed in the E-step, using the forward and backward variables as defined in Equation 6.100 and Equation 6.103. Because we are working in a hidden Markov model, we define the potentials as,

$$\psi_m(k, k') = \mathrm{p}_E(w_m \mid Y_m = k; \boldsymbol{\phi}) \Pr(Y_m = k \mid Y_{m-1} = k'; \lambda). \tag{6.114}$$

The expected counts are then,

$$E[\text{count}(W = i, Y = k)] = \sum_m \Pr(Y_m = k \mid \boldsymbol{w}_{1:M}) \delta(W_m = i) \tag{6.115}$$

$$= \sum_m \frac{\Pr(Y_m = k, \boldsymbol{w}_{1:m}) \mathrm{p}(\boldsymbol{w}_{m+1:M} \mid Y_m = k)}{\mathrm{p}(\boldsymbol{w}_{1:M})} \delta(w_m = i) \tag{6.116}$$

$$= \frac{1}{\alpha_M(\blacklozenge)} \sum_m \alpha_m(k) \beta_m(k) \delta(w_m = i) \tag{6.117}$$

We use the chain rule to separate $\boldsymbol{w}_{1:m}$ and $\boldsymbol{w}_{m+1:M}$, and then use the definitions of the forward and backward variables. In the final step, we normalize by $\mathrm{p}(\boldsymbol{w}_{1:M}) = \alpha_{M+1}(\blacklozenge) = \beta_0(\lozenge)$:

$$E[\text{count}(Y_m = k, Y_{m-1} = k')] = \sum_m P(Y_m = k, Y_{m-1} = k' \mid \boldsymbol{w}_{1:M}) \tag{6.118}$$

$$\propto \sum_m P(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) P(w_{m+1:M} \mid Y_m = k)$$
$$\times P(w_m, Y_m = k \mid Y_{m-1} = k') \tag{6.119}$$

$$= \sum_m P(Y_{m-1} = k', \boldsymbol{w}_{1:m-1}) P(w_{m+1:M} \mid Y_m = k)$$
$$\times \mathrm{p}(w_m \mid Y_m = k) P(Y_m = k \mid Y_{m-1} = k') \tag{6.120}$$

$$= \sum_m \alpha_{m-1}(k') \beta_m(k) \phi_{k, w_m} \lambda_{k' \to k}. \tag{6.121}$$

Again, we use the chain rule to separate out $\boldsymbol{w}_{1:m-1}$ and $\boldsymbol{w}_{m+1:M}$, and use the definitions of the forward and backward variables. The final computation also includes the parameters ϕ and λ, which govern (respectively) the emission and transition properties between w_m, y_m, and y_{m-1}. Note that the derivation only shows how to compute this to a constant of proportionality; we would divide by $\mathrm{p}(\boldsymbol{w}_{1:M})$ to go from the joint probability $P(Y_{m-1} = k', Y_m = k, \boldsymbol{w}_{1:M})$ to the desired conditional $\Pr(Y_{m-1} = k', Y_m = k \mid \boldsymbol{w}_{1:M})$.

Linear dynamical systems

The forward-backward algorithm can be viewed as Bayesian state estimation in a discrete state space. In a continuous state space, $y_m \in \mathbb{R}$, the equivalent algorithm is the **Kalman**

Smoother. It also computes marginals $\text{p}(y_m \mid \boldsymbol{x}_{1:M})$, using a similar two-step algorithm of forward and backward passes. Instead of computing a trellis of values at each step, we would compute a probability density function $q_{y_m}(y_m; \mu_m, \Sigma_m)$, characterized by a mean μ_m and a covariance Σ_m around the latent state. Connections between the Kalman Smoother and the forward-backward algorithm are elucidated by Minka (1999) and Murphy (2012).

Alternative unsupervised learning methods

As noted in section 4.4, expectation-maximization is just one of many techniques for structure induction. One alternative is to use a family of randomized algorithms called **Markov Chain Monte Carlo (MCMC)**. In these algorithms, we compute a marginal distribution over the latent variable $\boldsymbol{y}$ **empirically**, by drawing random samples. The randomness explains the "Monte Carlo" part of the name; typically, we employ a Markov Chain sampling procedure, meaning that each sample is drawn from a distribution that depends only on the previous sample (and not on the entire sampling history). A simple MCMC algorithm is **Gibbs Sampling**, in which we iteratively sample each y_m conditioned on all the others (Finkel et al., 2005):

$$\text{p}(y_m \mid \boldsymbol{y}_{-m}, \boldsymbol{w}_{1:M}) \propto \text{p}(w_m \mid y_m)\text{p}(y_m \mid \boldsymbol{y}_{-m}). \tag{6.122}$$

Gibbs Sampling has been applied to unsupervised part-of-speech tagging by Goldwater and Griffiths (2007). *Beam sampling* is a more sophisticated sampling algorithm, which randomly draws entire sequences $\boldsymbol{y}_{1:M}$, rather than individual tags y_m; this algorithm was applied to unsupervised part-of-speech tagging by Van Gael et al. (2009).

EM is guaranteed to find only a local optimum; MCMC algorithms will converge to the true posterior distribution $\text{p}(\boldsymbol{y}_{1:M} \mid \boldsymbol{w}_{1:M})$, but this is only guaranteed in the limit of infinite samples. Recent work has explored the use of **spectral learning** for latent variable models, which use matrix and tensor decompositions to provide guaranteed convergence under mild assumptions (Song et al., 2010; Hsu et al., 2012).

Exercises

1. Consider the garden path sentence, *The old man the boat.* Given word-tag and tag-tag features, what inequality in the weights must hold for the correct tag sequence to outscore the garden path tag sequence for this example?

2. Sketch out an algorithm for a variant of Viterbi that returns the top-k sequences. What is the time and space complexity of this algorithm?

3. Show how to compute the marginal probability $\Pr(Y_{m-2} = k, Y_m = k')$, in terms of the forwards and backward variables, and the potentials ψ_m.

Chapter 7

Applications of sequence labeling

Sequence labeling has applications throughout computational linguistics. This chapter focuses on the classical applications of part-of-speech tagging, shallow parsing, and named entity recognition, and touches briefly on two applications to interactive settings: dialogue act recognition and the detection of code-switching points between languages.

7.1 Part-of-speech tagging

The **syntax** of a language is the collection of principles under which sequences of words are judged to be grammatically acceptable by fluent speakers. One of the most basic syntactic concepts is the **part-of-speech** (POS), which refers to the syntactic role of each word in a sentence. We have already referred to this concept informally in the previous chapter, and you may have some intuitions from your own study of English. For example, in the sentence *Akanksha likes vegetarian sandwiches*, you may already know that *Akanksha* and *sandwiches* are nouns, *likes* is a verb, and *vegetarian* is an adjective.

Parts-of-speech can help to disentangle or explain various linguistic problems. Recall Chomsky's proposed distinction in chapter 5:

(7.1) Colorless green ideas sleep furiously.

(7.2) *Ideas colorless furiously green sleep.

Why is the first grammatical and the second not? One explanation is that the first example contains part-of-speech transitions that are typical in English: adjective to adjective, adjective to noun, noun to verb, and verb to adverb. In contrast, the second sentence contains transitions that are unusual: noun to adjective and adjective to verb. The ambiguity in sentences like *teacher strikes idle children* can also be explained in terms of parts of speech: in the interpretation that was likely intended, *strikes* is a noun and *idle* is a verb; in the alternative explanation, *strikes* is a verb and *idle* is an adjective.

Part-of-speech tagging is often taken as a early step in a natural language processing pipeline. Indeed, parts-of-speech provide features that can be useful for many of the tasks that we will encounter later, such as parsing, coreference resolution, and relation extraction. [todo: say more here]

Part-of-speech inventories

We have discussed a few parts-of-speech already: noun, verb, adjective, adverb. Jurafsky and Martin (2009) describe these as the four major **open word classes** , meaning that these are classes in which new words can be created.[1] These four open classes can be divided into more fine-grained subcategories, such as proper and common nouns; in addition, there are a number of closed classes. This section provides an overview of part-of-speech categories for English, but see Bender (2013) for a deeper linguistic perspective.

When creating a part-of-speech tagged dataset, the tagset inventory must be explicitly defined. The best known POS dataset for English is the Penn Treebank (PTB; Marcus et al., 1993), which includes a set of 45 tags. This chapter describes the linguistics of part-of-speech categories, and lists the corresponding PTB tags. The next section contrasts the PTB tagset with other tagsets for English and for other languages.

- **Nouns** describe entities and concepts.
  - **Proper nouns** name specific people and entities: e.g., *Georgia Tech*, *Janet*, *Buddhism*. In English, proper nouns are usually capitalized. The Penn Treebank (PTB) tags for proper nouns are: NNP (singular), NNPS (plural).
  - **Common nouns** cover all other nouns. In English, they are often preceded by determiners, e.g. *the book*, *a university*, *some people*. Common nouns decompose into two main types:
    - **Count nouns** have a plural and need an article in the singular, *dogs*, *the dog*;
    - **Mass nouns** don't have a plural and don't need an article in the singular:

      (7.3) ***Fire*** *is dangerous.*

      (7.4) ***Skiing*** *is an expensive hobby.*

    In the Penn Treebank, singular and mass nouns are tagged NN, and plural nouns are tagged NNS.
  - **Pronouns** refer to specific entities or events that are already known to the reader or listener.

[1] Languages need not have all four classes: for example, it has been argued that Korean does not have adjectives Kim (2002).

  - **Personal pronouns** refer to people or entities: *you, she, I, it, me*. The PTB tag is PRP.
  - **Possessive pronouns** are pronouns that indicate possession: *your, her, my, its, one's, our*. The PTB tag is PRP$.
  - **Wh-pronouns** (WP) are used in question forms, and as relative pronouns:

    (7.5) ***Where** are you going?*

    (7.6) *The man **who** wasn't there.*

    Possessive wh-pronouns (e.g., *The guy **whose** email got hacked*) are distinguished in the PTB with the tag WP$.

  Pronouns are considered a **closed class** , because unlike other nouns, it is generally not possible to introduce new pronouns.[2]

- **Verbs** describe activities, processes, and events. For example, *eat, write, sleep* are all verbs.
  - Just as nouns can be differentiated by number, verbs are differentiated by properties of the action they describe, and by their form. For English, the Penn Treebank differentiates the following types of verbs: VB (infinitive, e.g. *to shake*), VBD (past, e.g. *shook*), VBG (present participle, e.g. *shaking*), VBN (past participle, e.g. *shaken*), VBZ (present third-person singular, e.g. *shakes*), VBP (present, non-third-person singular, e.g. *shake*).

    Note that these verb classes include properties of the event like tense, as well as subject-verb agreement (third-person singular). Many languages have much more complex inflectional systems than English. For example, French verbs have unique inflections for every combination of person and number; when combined with features such as tense, mood, and aspect, there are several dozen possible inflections for each verb.
  - **Modals** are a closed subclass of verbs; they give additional information about the event described by the main verb of the sentence, e.g., *someone **should** do something*. In the PTB, their tag is MD. The PTB distinguishes modal verbs as all verbs which do not take a *-s* suffix in the third person singular present (Santorini, 1990).
  - The verb *to be* requires special treatment, as it must appear with a **predicative adjective** or noun, e.g.

    (7.7) *She is hungry.*

[2] Recent efforts to create gender-neutral singular pronouns in English are the exception that proves the rule: while battles over *s/he* and singular *they* have raged for decades, legions of new common nouns have been introduced (e.g., *selfie*, *emoji*) without much controversy.