CHAPTER

5 Logistic Regression

"And how do you know that these fine begonias are not of equal importance?"
Hercule Poirot, in Agatha Christie's *The Mysterious Affair at Styles*

Detective stories are as littered with clues as texts are with words. Yet for the poor reader it can be challenging to know how to weigh the author's clues in order to make the crucial classification task: deciding whodunnit.

logistic regression

In this chapter we introduce an algorithm that is admirably suited for discovering the link between features or clues and some particular outcome: **logistic regression**. Indeed, logistic regression is one of the most important analytic tools in the social and natural sciences. In natural language processing, logistic regression is the baseline supervised machine learning algorithm for classification, and also has a very close relationship with neural networks. As we will see in Chapter 7, a neural network can be viewed as a series of logistic regression classifiers stacked on top of each other. Thus the classification and machine learning techniques introduced here will play an important role throughout the book.

Logistic regression can be used to classify an observation into one of two classes (like 'positive sentiment' and 'negative sentiment'), or into one of many classes. Because the mathematics for the two-class case is simpler, we'll describe this special case of logistic regression first in the next few sections, and then briefly summarize the use of **multinomial logistic regression** for more than two classes in Section 5.3.

We'll introduce the mathematics of logistic regression in the next few sections. But let's begin with some high-level issues.

Generative and Discriminative Classifiers: The most important difference between naive Bayes and logistic regression is that logistic regression is a **discriminative** classifier while naive Bayes is a **generative** classifier.

These are two very different frameworks for how to build a machine learning model. Consider a visual metaphor: imagine we're trying to distinguish dog images from cat images. A generative model would have the goal of understanding what dogs look like and what cats look like. You might literally ask such a model to 'generate', i.e., draw, a dog. Given a test image, the system then asks whether it's the cat model or the dog model that better fits (is less surprised by) the image, and chooses that as its label.

A discriminative model, by contrast, is only trying to learn to distinguish the classes (perhaps without learning much about them). So maybe all the dogs in the training data are wearing collars and the cats aren't. If that one feature neatly separates the classes, the model is satisfied. If you ask such a model what it knows about cats all it can say is that they don't wear collars.

discriminative model learns what distinguishes the species rather than actually learning anything about them.

More formally, recall that the naive Bayes assigns a class c to a document d not by directly computing $P(c|d)$ but by computing a likelihood and a prior

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \tag{5.1}$$

generative model

A **generative model** like naive Bayes makes use of this **likelihood** term, which expresses how to generate the features of a document *if we knew it was of class c*.

discriminative model

By contrast a **discriminative model** in this text categorization scenario attempts to **directly** compute $P(c|d)$. Perhaps it will learn to assign a high weight to document features that directly improve its ability to *discriminate* between possible classes, even if it couldn't generate an example of one of the classes.

Components of a probabilistic machine learning classifier: Like naive Bayes, logistic regression is a probabilistic classifier that makes use of supervised machine learning. Machine learning classifiers require a training corpus of m input/output pairs $(x^{(i)}, y^{(i)})$. (We'll use superscripts in parentheses to refer to individual instances in the training set—for sentiment classification each instance might be an individual document to be classified.) A machine learning system for classification then has four components:

1. A **feature representation** of the input. For each input observation $x^{(i)}$, this will be a vector of features $[x_1, x_2, ..., x_n]$. We will generally refer to feature i for input $x^{(j)}$ as $x_i^{(j)}$, sometimes simplified as x_i, but we will also see the notation f_i, $f_i(x)$, or, for multiclass classification, $f_i(c, x)$.
2. A classification function that computes $\hat{y}$, the estimated class, via $p(y|x)$. In the next section we will introduce the **sigmoid** and **softmax** tools for classification.
3. An objective function that we want to optimize for learning, usually involving minimizing a loss function corresponding to error on training examples. We will introduce the **cross-entropy loss function**.
4. An algorithm for optimizing the objective function. We introduce the **stochastic gradient descent** algorithm.

Logistic regression has two phases:

training: We train the system (specifically the weights w and b, introduced below) using stochastic gradient descent and the cross-entropy loss.

test: Given a test example x we compute $p(y|x)$ and return the higher probability label $y = 1$ or $y = 0$.

5.1 The sigmoid function

The goal of binary logistic regression is to train a classifier that can make a binary decision about the class of a new input observation. Here we introduce the **sigmoid** classifier that will help us make this decision.

Consider a single input observation x, which we will represent by a vector of features $[x_1, x_2, ..., x_n]$. (We'll show sample features in the next subsection.) The classifier output y can be 1 (meaning the observation is a member of the class) or 0 (the observation is not a member of the class). We want to know the probability

$P(y=1|x)$ that this observation is a member of the class. So perhaps the decision is "positive sentiment" versus "negative sentiment", the features represent counts of words in a document, $P(y=1|x)$ is the probability that the document has positive sentiment, and $P(y=0|x)$ is the probability that the document has negative sentiment.

Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**. Each weight w_i is a real number, and is associated with one of the input features x_i. The weight w_i represents how important that input feature is to the classification decision, and can be positive (providing evidence that the instance being classified belongs in the positive class) or negative (providing evidence that the instance being classified belongs in the negative class). Thus we might expect in a sentiment task the word *awesome* to have a high positive weight, and *abysmal* to have a very negative weight. The **bias term**, also called the **intercept**, is another real number that's added to the weighted inputs.

bias term
intercept

To make a decision on a test instance—after we've learned the weights in training—the classifier first multiplies each x_i by its weight w_i, sums up the weighted features, and adds the bias term b. The resulting single number z expresses the weighted sum of the evidence for the class.

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b \tag{5.2}$$

dot product

In the rest of the book we'll represent such sums using the **dot product** notation from linear algebra. The dot product of two vectors $\mathbf{a}$ and $\mathbf{b}$, written as $\mathbf{a}\cdot\mathbf{b}$, is the sum of the products of the corresponding elements of each vector. (Notice that we represent vectors using the boldface notation $\mathbf{b}$). Thus the following is an equivalent formation to Eq. 5.2:

$$z = \mathbf{w}\cdot\mathbf{x} + b \tag{5.3}$$

But note that nothing in Eq. 5.3 forces z to be a legal probability, that is, to lie between 0 and 1. In fact, since weights are real-valued, the output might even be negative; z ranges from $-\infty$ to ∞.

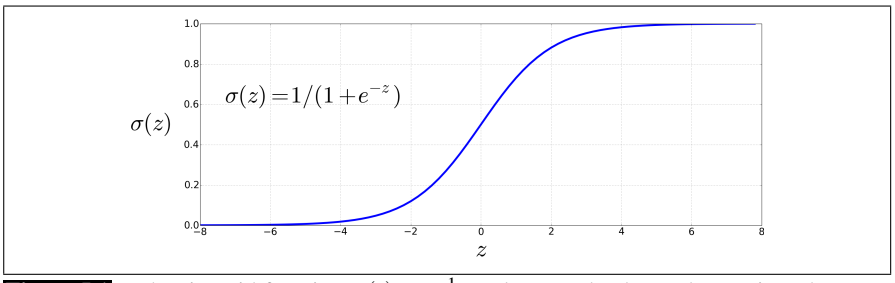

Figure 5.1 The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

sigmoid
logistic function

To create a probability, we'll pass z through the **sigmoid** function, $\sigma(z)$. The sigmoid function (named because it looks like an s) is also called the **logistic function**, and gives logistic regression its name. The sigmoid has the following equation, shown graphically in Fig. 5.1:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-z)} \tag{5.4}$$

(For the rest of the book, we'll use the notation $\exp(x)$ to mean e^x.) The sigmoid has a number of advantages; it takes a real-valued number and maps it into the range $(0,1)$, which is just what we want for a probability. Because it is nearly linear around 0 but flattens toward the ends, it tends to squash outlier values toward 0 or 1. And it's differentiable, which as we'll see in Section 5.10 will be handy for learning.

We're almost there. If we apply the sigmoid to the sum of the weighted features, we get a number between 0 and 1. To make it a probability, we just need to make sure that the two cases, $p(y=1)$ and $p(y=0)$, sum to 1. We can do this as follows:

$$\begin{aligned} P(y=1) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ P(y=0) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= 1-\frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ &= \frac{\exp(-(\mathbf{w}\cdot\mathbf{x}+b))}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \end{aligned} \tag{5.5}$$

The sigmoid function has the property

$$1-\sigma(x)=\sigma(-x) \tag{5.6}$$

so we could also have expressed $P(y=0)$ as $\sigma(-(\mathbf{w}\cdot\mathbf{x}+b))$.

Finally, one terminological point. The input to the sigmoid function, the score $z=\mathbf{w}\cdot\mathbf{x}+b$ from Eq. 5.3, is often called the **logit**. This is because the logit function is the inverse of the sigmoid. The logit function is the log of the odds ratio $\frac{p}{1-p}$:

logit

$$\text{logit}(p)=\sigma^{-1}(p)=\ln\frac{p}{1-p} \tag{5.7}$$

Using the term **logit** for z is a way of reminding us that by using the sigmoid to turn z (which ranges from $-\infty$ to ∞) into a probability, we are implicitly interpreting z as not just any real-valued number, but as specifically a log odds.

5.2 Classification with Logistic Regression

The sigmoid function from the prior section thus gives us a way to take an instance x and compute the probability $P(y=1|x)$.

How do we make a decision about which class to apply to a test instance x? For a given x, we say yes if the probability $P(y=1|x)$ is more than .5, and no otherwise. We call .5 the **decision boundary**:

decision boundary

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y=1|x)>0.5 \\ 0 & \text{otherwise} \end{cases}$$

Let's have some examples of applying logistic regression as a classifier for language tasks.

5.2.1 Sentiment Classification

Suppose we are doing binary sentiment classification on movie review text, and we would like to know whether to assign the sentiment class + or − to a review document *doc*. We'll represent each input observation by the 6 features $x_1 \ldots x_6$ of the input shown in the following table; Fig. 5.2 shows the features in a sample mini test document.

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon words ∈ doc)	3
x_2	count(negative lexicon words ∈ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns ∈ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	ln(word count of doc)	$\ln(66) = 4.19$

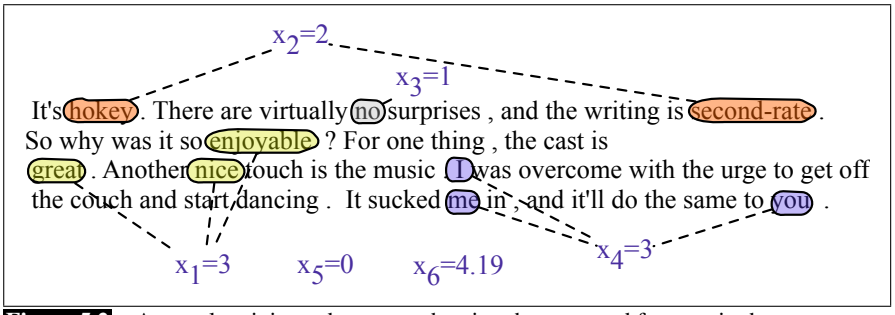

Figure 5.2 A sample mini test document showing the extracted features in the vector x.

Let's assume for the moment that we've already learned a real-valued weight for each of these features, and that the 6 weights corresponding to the 6 features are $[2.5, -5.0, -1.2, 0.5, 2.0, 0.7]$, while $b = 0.1$. (We'll discuss in the next section how the weights are learned.) The weight w_1, for example indicates how important a feature the number of positive lexicon words (*great*, *nice*, *enjoyable*, etc.) is to a positive sentiment decision, while w_2 tells us the importance of negative lexicon words. Note that $w_1 = 2.5$ is positive, while $w_2 = -5.0$, meaning that negative words are negatively associated with a positive sentiment decision, and are about twice as important as positive words.

Given these 6 features and the input review x, $P(+|x)$ and $P(-|x)$ can be computed using Eq. 5.5:

$$\begin{aligned} p(+|x) = P(y=1|x) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \sigma([2.5,-5.0,-1.2,0.5,2.0,0.7]\cdot[3,2,1,3,0,4.19]+0.1) \\ &= \sigma(.833) \\ &= 0.70 \qquad (5.8) \\ p(-|x) = P(y=0|x) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= 0.30 \end{aligned}$$

5.2.2 Other classification tasks and features

period disambiguation

Logistic regression is applied to all sorts of NLP tasks, and any property of the input can be a feature. Consider the task of **period disambiguation**: deciding if a period is the end of a sentence or part of a word, by classifying each period into one of two classes, EOS (end-of-sentence) and not-EOS. We might use features like x_1 below expressing that the current word is lower case, perhaps with a positive weight. Or a feature expressing that the current word is in our abbreviations dictionary ("Prof."), perhaps with a negative weight. A feature can also express a combination of properties. For example a period following an upper case word is likely to be an EOS, but if the word itself is *St.* and the previous word is capitalized then the period is likely part of a shortening of the word *street* following a street name.

$$x_1 = \begin{cases} 1 & \text{if } \text{“}Case(w_i) = \text{Lower”} \\ 0 & \text{otherwise} \end{cases}$$

$$x_2 = \begin{cases} 1 & \text{if } \text{“}w_i \in \text{AcronymDict”} \\ 0 & \text{otherwise} \end{cases}$$

$$x_3 = \begin{cases} 1 & \text{if } \text{“}w_i = \text{St. \& } Case(w_{i-1}) = \text{Upper”} \\ 0 & \text{otherwise} \end{cases}$$

Designing versus learning features: In classic models, features are designed by hand by examining the training set with an eye to linguistic intuitions and literature, supplemented by insights from error analysis on the training set of an early version of a system. We can also consider (**feature interactions**), complex features that are combinations of more primitive features. We saw such a feature for period disambiguation above, where a period on the word *St.* was less likely to be the end of the sentence if the previous word was capitalized. Features can be created automatically via **feature templates**, abstract specifications of features. For example a bigram template for period disambiguation might create a feature for every pair of words that occurs before a period in the training set. Thus the feature space is sparse, since we only have to create a feature if that n-gram exists in that position in the training set. The feature is generally created as a hash from the string descriptions. A user description of a feature as, "bigram(American breakfast)" is hashed into a unique integer i that becomes the feature number f_i.

feature interactions

feature templates

It should be clear from the prior paragraph that designing features by hand requires extensive human effort. For this reason, recent NLP systems avoid hand-designed features and instead focus on **representation learning**: ways to learn features automatically in an unsupervised way from the input. We'll introduce methods for representation learning in Chapter 6 and Chapter 7.

Scaling input features: When different input features have extremely different ranges of values, it's common to rescale them so they have comparable ranges. We **standardize** input values by centering them to result in a zero mean and a standard deviation of one (this transformation is sometimes called the **z-score**). That is, if μ_i is the mean of the values of feature x_i across the m observations in the input dataset, and σ_i is the standard deviation of the values of features x_i across the input dataset, we can replace each feature x_i by a new feature x'_i computed as follows:

standardize

z-score

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \qquad \sigma_i = \sqrt{\frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2}$$

$$x'_i = \frac{x_i - \mu_i}{\sigma_i} \tag{5.9}$$

normalize

Alternatively, we can **normalize** the input features values to lie between 0 and 1:

$$x_i' = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)} \tag{5.10}$$

Having input data with comparable range is useful when comparing values across features. Data scaling is especially important in large neural networks, since it helps speed up gradient descent.

5.2.3 Processing many examples at once

We've shown the equations for logistic regression for a single example. But in practice we'll of course want to process an entire test set with many examples. Let's suppose we have a test set consisting of m test examples each of which we'd like to classify. We'll continue to use the notation from page 2, in which a superscript value in parentheses refers to the example index in some set of data (either for training or for test). So in this case each test example $x^{(i)}$ has a feature vector $\mathbf{x}^{(i)}$, $1 \leq i \leq m$. (As usual, we'll represent vectors and matrices in bold.)

One way to compute each output value $\hat{y}^{(i)}$ is just to have a for-loop, and compute each test example one at a time:

$$\begin{aligned} \textbf{foreach} \quad & x^{(i)} \text{ in input } [x^{(1)}, x^{(2)}, ..., x^{(m)}] \\ & y^{(i)} = \sigma(\mathbf{w} \cdot \mathbf{x}^{(\mathbf{i})} + b) \end{aligned} \tag{5.11}$$

For the first 3 test examples, then, we would be separately computing the predicted $\hat{y}^{(i)}$ as follows:

$$\begin{aligned} P(y^{(1)} = 1 | x^{(1)}) &= \sigma(\mathbf{w} \cdot \mathbf{x}^{(\mathbf{1})} + b) \\ P(y^{(2)} = 1 | x^{(2)}) &= \sigma(\mathbf{w} \cdot \mathbf{x}^{(\mathbf{2})} + b) \\ P(y^{(3)} = 1 | x^{(3)}) &= \sigma(\mathbf{w} \cdot \mathbf{x}^{(\mathbf{3})} + b) \end{aligned}$$

But it turns out that we can slightly modify our original equation Eq. 5.5 to do this much more efficiently. We'll use matrix arithmetic to assign a class to all the examples with one matrix operation!

First, we'll pack all the input feature vectors for each input x into a single input matrix $\mathbf{X}$, where each row i is a row vector consisting of the feature vector for input example $x^{(i)}$ (i.e., the vector $\mathbf{x}^{(i)}$). Assuming each example has f features and weights, $\mathbf{X}$ will therefore be a matrix of shape $[m \times f]$, as follows:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_f^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_f^{(2)} \\ x_1^{(3)} & x_2^{(3)} & \dots & x_f^{(3)} \\ \dots & & & \end{bmatrix} \tag{5.12}$$

Now if we introduce $\mathbf{b}$ as a vector of length m which consists of the scalar bias term b repeated m times, $\mathbf{b} = [b, b, ..., b]$, and $\hat{\mathbf{y}} = [\hat{y}^{(1)}, \hat{y}^{(2)} ..., \hat{y}^{(m)}]$ as the vector of outputs (one scalar $\hat{y}^{(i)}$ for each input $x^{(i)}$ and its feature vector $\mathbf{x}^{(i)}$), and represent the weight vector $\mathbf{w}$ as a column vector, we can compute all the outputs with a single matrix multiplication and one addition:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{b} \tag{5.13}$$

You should convince yourself that Eq. 5.13 computes the same thing as our for-loop in Eq. 5.11. For example $\hat{y}^{(1)}$, the first entry of the output vector $\mathbf{y}$, will correctly be:

$$\hat{y}^{(1)} = [x_1^{(1)}, x_2^{(1)}, ..., x_f^{(1)}] \cdot [w_1, w_2, ..., w_f] + b \tag{5.14}$$

Note that we had to reorder $\mathbf{X}$ and $\mathbf{w}$ from the order they appeared in in Eq. 5.5 to make the multiplications come out properly. Here is Eq. 5.13 again with the shapes shown:

$$\begin{array}{ccccccc} \mathbf{y} & = & \mathbf{X} & & \mathbf{w} & + & \mathbf{b} \\ (m \times 1) & & (m \times f) & & (f \times 1) & & (m \times 1) \end{array} \tag{5.15}$$

Modern compilers and compute hardware can compute this matrix operation very efficiently, making the computation much faster, which becomes important when training or testing on very large datasets.

Note by the way that we could have kept $\mathbf{X}$ and $\mathbf{w}$ in the original order ($\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{b}$) if we had chosen to define $\mathbf{X}$ differently as a matrix of column vectors, one vector for each input example, instead of row vectors, and then it would have shape $[f \times m]$. But we conventionally represent inputs as rows.

5.2.4 Choosing a classifier

Logistic regression has a number of advantages over naive Bayes. Naive Bayes has overly strong conditional independence assumptions. Consider two features which are strongly correlated; in fact, imagine that we just add the same feature f_1 twice. Naive Bayes will treat both copies of f_1 as if they were separate, multiplying them both in, overestimating the evidence. By contrast, logistic regression is much more robust to correlated features; if two features f_1 and f_2 are perfectly correlated, regression will simply assign part of the weight to w_1 and part to w_2. Thus when there are many correlated features, logistic regression will assign a more accurate probability than naive Bayes. So logistic regression generally works better on larger documents or datasets and is a common default.

Despite the less accurate probabilities, naive Bayes still often makes the correct classification decision. Furthermore, naive Bayes can work extremely well (sometimes even better than logistic regression) on very small datasets (Ng and Jordan, 2002) or short documents (Wang and Manning, 2012). Furthermore, naive Bayes is easy to implement and very fast to train (there's no optimization step). So it's still a reasonable approach to use in some situations.

5.3 Multinomial logistic regression

Sometimes we need more than two classes. Perhaps we might want to do 3-way sentiment classification (positive, negative, or neutral). Or we could be assigning some of the labels we will introduce in Chapter 17, like the part of speech of a word (choosing from 10, 30, or even 50 different parts of speech), or the named entity type of a phrase (choosing from tags like person, location, organization).

multinomial logistic regression

In such cases we use **multinomial logistic regression**, also called **softmax regression** (in older NLP literature you will sometimes see the name **maxent classifier**). In multinomial logistic regression we want to label each observation with a class k from a set of K classes, under the stipulation that only one of these classes is

the correct one (sometimes called **hard classification**; an observation can not be in multiple classes). Let's use the following representation: the output $\mathbf{y}$ for each input $\mathbf{x}$ will be a vector of length K. If class c is the correct class, we'll set $y_c = 1$, and set all the other elements of $\mathbf{y}$ to be 0, i.e., $y_c = 1$ and $y_j = 0 \;\; \forall j \neq c$. A vector like this $\mathbf{y}$, with one value=1 and the rest 0, is called a **one-hot vector**. The job of the classifier is to produce an estimate vector $\hat{\mathbf{y}}$. For each class k, the value $\hat{y}_k$ will be the classifier's estimate of the probability $p(y_k = 1|\mathbf{x})$.

5.3.1 Softmax

softmax

The multinomial logistic classifier uses a generalization of the sigmoid, called the **softmax** function, to compute $p(y_k = 1|\mathbf{x})$. The softmax function takes a vector $\mathbf{z} = [z_1, z_2, ..., z_K]$ of K arbitrary values and maps them to a probability distribution, with each value in the range [0,1], and all the values summing to 1. Like the sigmoid, it is an exponential function.

For a vector $\mathbf{z}$ of dimensionality K, the softmax is defined as:

$$\text{softmax}(z_i) \;=\; \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \quad 1 \leq i \leq K \tag{5.16}$$

The softmax of an input vector $\mathbf{z} = [z_1, z_2, ..., z_K]$ is thus a vector itself:

$$\text{softmax}(\mathbf{z}) \;=\; \left[\frac{\exp(z_1)}{\sum_{i=1}^{K} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K} \exp(z_i)}, ..., \frac{\exp(z_K)}{\sum_{i=1}^{K} \exp(z_i)} \right] \tag{5.17}$$

The denominator $\sum_{i=1}^{K} \exp(\mathbf{z}_i)$ is used to normalize all the values into probabilities. Thus for example given a vector:

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

the resulting (rounded) softmax($\mathbf{z}$) is

$$[0.05, 0.09, 0.01, 0.1, 0.74, 0.01]$$

Like the sigmoid, the softmax has the property of squashing values toward 0 or 1. Thus if one of the inputs is larger than the others, it will tend to push its probability toward 1, and suppress the probabilities of the smaller inputs.

Finally, note that, just as for the sigmoid, we refer to $\mathbf{z}$, the vector of scores that is the input to the softmax, as **logits** (see Eq. 5.7).

5.3.2 Applying softmax in logistic regression

When we apply softmax for logistic regression, the input will (just as for the sigmoid) be the dot product between a weight vector $\mathbf{w}$ and an input vector $\mathbf{x}$ (plus a bias). But now we'll need separate weight vectors $\mathbf{w}_k$ and bias b_k for each of the K classes. The probability of each of our output classes $\hat{y}_k$ can thus be computed as:

$$p(y_k = 1|\mathbf{x}) \;=\; \frac{\exp(\mathbf{w}_k \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + b_j)} \tag{5.18}$$

The form of Eq. 5.18 makes it seem that we would compute each output separately. Instead, it's more common to set up the equation for more efficient computation by modern vector processing hardware. We'll do this by representing the set of K weight vectors as a weight matrix W and a bias vector $\mathbf{b}$. Each row k of $\mathbf{W}$ corresponds to the vector of weights w_k. $\mathbf{W}$ thus has shape $[K \times f]$, for K the number of output classes and f the number of input features. The bias vector $\mathbf{b}$ has one value for each of the K output classes. If we represent the weights in this way, we can compute $\hat{\mathbf{y}}$, the vector of output probabilities for each of the K classes, by a single elegant equation:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{5.19}$$

If you work out the matrix arithmetic, you can see that the estimated score of the first output class $\hat{y}_1$ (before we take the softmax) will correctly turn out to be $\mathbf{w}_1 \cdot \mathbf{x} + b_1$.

prototype

One helpful interpretation of the weight matrix $\mathbf{W}$ is to see each row $\mathbf{w}_k$ as a **prototype** of class k. The weight vector $\mathbf{w}_k$ that is learned represents the class as a kind of template. Since two vectors that are more similar to each other have a higher dot product with each other, the dot product acts as a similarity function. Logistic regression is thus learning an **exemplar** representation for each class, such that incoming vectors are assigned the class k they are most similar to from the K classes.

Fig. 5.3 shows the difference between binary and multinomial logistic regression by illustrating the weight vector versus weight matrix in the computation of the output class probabilities.

5.3.3 Features in Multinomial Logistic Regression

Features in multinomial logistic regression act like features in binary logistic regression, with the difference mentioned above that we'll need separate weight vectors and biases for each of the K classes. Recall our binary exclamation point feature x_5 from page 5:

$$x_5 = \begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$$

In binary classification a positive weight w_5 on a feature influences the classifier toward $y = 1$ (positive sentiment) and a negative weight influences it toward $y = 0$ (negative sentiment) with the absolute value indicating how important the feature is. For multinomial logistic regression, by contrast, with separate weights for each class, a feature can be evidence for or against each individual class.

In 3-way multiclass sentiment classification, for example, we must assign each document one of the 3 classes $+$, $-$, or 0 (neutral). Now a feature related to exclamation marks might have a negative weight for 0 documents, and a positive weight for $+$ or $-$ documents:

Feature	Definition	$w_{5,+}$	$w_{5,-}$	$w_{5,0}$
$f_5(x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	3.5	3.1	-5.3

Because these feature weights are dependent both on the input text and the output class, we sometimes make this dependence explicit and represent the features themselves as $f(x, y)$: a function of both the input and the class. Using such a notation

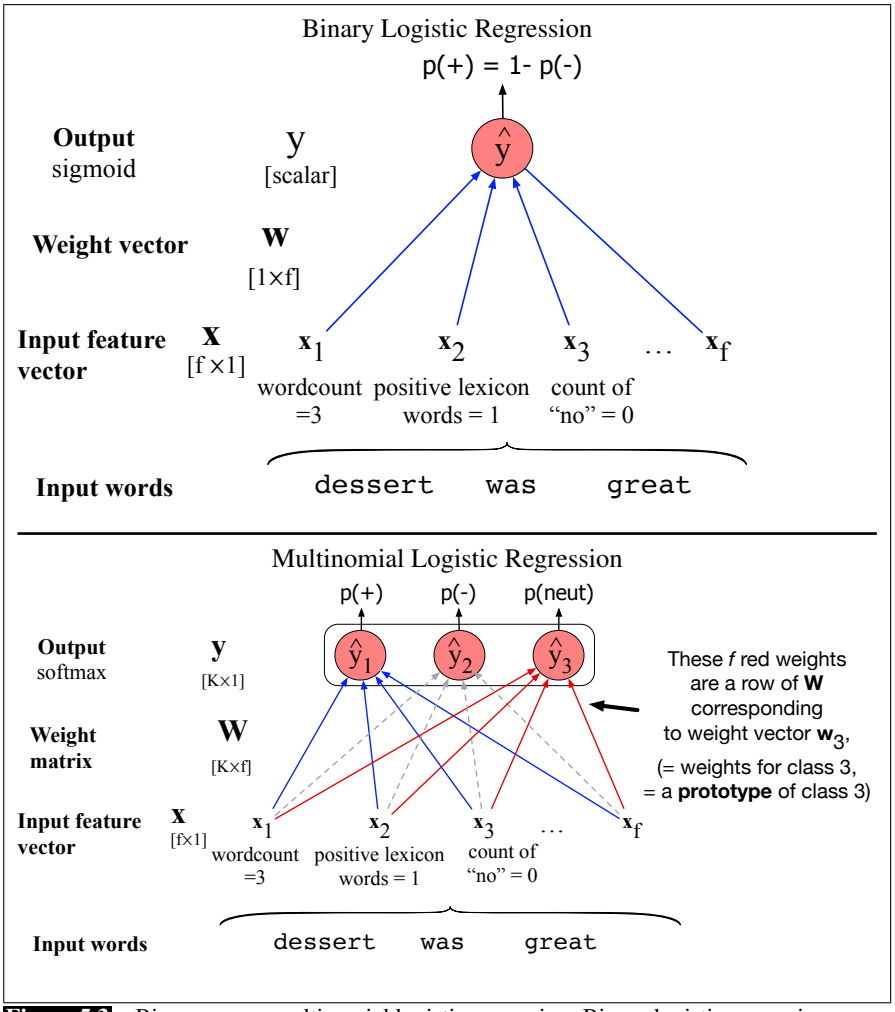

Figure 5.3 Binary versus multinomial logistic regression. Binary logistic regression uses a single weight vector $\mathbf{w}$, and has a scalar output $\hat{y}$. In multinomial logistic regression we have K separate weight vectors corresponding to the K classes, all packed into a single weight matrix $\mathbf{W}$, and a vector output $\hat{\mathbf{y}}$. We omit the biases from both figures for clarity.

$f_5(x)$ above could be represented as three features $f_5(x,+)$, $f_5(x,-)$, and $f_5(x,0)$, each of which has a single weight. We'll use this kind of notation in our description of the CRF in Chapter 17.

5.4 Learning in Logistic Regression

How are the parameters of the model, the weights $\mathbf{w}$ and bias b, learned? Logistic regression is an instance of supervised classification in which we know the correct label y (either 0 or 1) for each observation x. What the system produces via Eq. 5.5 is $\hat{y}$, the system's estimate of the true y. We want to learn parameters (meaning $\mathbf{w}$ and b) that make $\hat{y}$ for each training observation as close as possible to the true y.

This requires two components that we foreshadowed in the introduction to the chapter. The first is a metric for how close the current label ($\hat{y}$) is to the true gold

label y. Rather than measure similarity, we usually talk about the opposite of this: the *distance* between the system output and the gold output, and we call this distance the **loss** function or the **cost function**. In the next section we'll introduce the loss function that is commonly used for logistic regression and also for neural networks, the **cross-entropy loss**.

loss

The second thing we need is an optimization algorithm for iteratively updating the weights so as to minimize this loss function. The standard algorithm for this is **gradient descent**; we'll introduce the **stochastic gradient descent** algorithm in the following section.

We'll describe these algorithms for the simpler case of binary logistic regression in the next two sections, and then turn to multinomial logistic regression in Section 5.8.

5.5 The cross-entropy loss function

We need a loss function that expresses, for an observation x, how close the classifier output ($\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$) is to the correct output (y, which is 0 or 1). We'll call this:

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y \tag{5.20}$$

We do this via a loss function that prefers the correct class labels of the training examples to be *more likely*. This is called **conditional maximum likelihood estimation**: we choose the parameters w, b that **maximize the log probability of the true y labels in the training data** given the observations x. The resulting loss function is the **negative log likelihood loss**, generally called the **cross-entropy loss**.

cross-entropy loss

Let's derive this loss function, applied to a single observation x. We'd like to learn weights that maximize the probability of the correct label $p(y|x)$. Since there are only two discrete outcomes (1 or 0), this is a Bernoulli distribution, and we can express the probability $p(y|x)$ that our classifier produces for one observation as the following (keeping in mind that if $y = 1$, Eq. 5.21 simplifies to $\hat{y}$; if $y = 0$, Eq. 5.21 simplifies to $1 - \hat{y}$):

$$p(y|x) = \hat{y}^y (1-\hat{y})^{1-y} \tag{5.21}$$

Now we take the log of both sides. This will turn out to be handy mathematically, and doesn't hurt us; whatever values maximize a probability will also maximize the log of the probability:

$$\begin{aligned} \log p(y|x) &= \log \left[\hat{y}^y (1-\hat{y})^{1-y}\right] \\ &= y \log \hat{y} + (1-y) \log(1-\hat{y}) \end{aligned} \tag{5.22}$$

Eq. 5.22 describes a log likelihood that should be maximized. In order to turn this into a loss function (something that we need to minimize), we'll just flip the sign on Eq. 5.22. The result is the cross-entropy loss L_{CE}:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})] \tag{5.23}$$

Finally, we can plug in the definition of $\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$:

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \tag{5.24}$$

Let's see if this loss function does the right thing for our example from Fig. 5.2. We want the loss to be smaller if the model's estimate is close to correct, and bigger if the model is confused. So first let's suppose the correct gold label for the sentiment example in Fig. 5.2 is positive, i.e., $y = 1$. In this case our model is doing well, since from Eq. 5.8 it indeed gave the example a higher probability of being positive (.70) than negative (.30). If we plug $\sigma(\mathbf{w} \cdot \mathbf{x} + b) = .70$ and $y = 1$ into Eq. 5.24, the right side of the equation drops out, leading to the following loss (we'll use log to mean natural log when the base is not specified):

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log \sigma(\mathbf{w} \cdot \mathbf{x} + b)] \\ &= -\log(.70) \\ &= .36 \end{aligned}$$

By contrast, let's pretend instead that the example in Fig. 5.2 was actually negative, i.e., $y = 0$ (perhaps the reviewer went on to say "But bottom line, the movie is terrible! I beg you not to see it!"). In this case our model is confused and we'd want the loss to be higher. Now if we plug $y = 0$ and $1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b) = .30$ from Eq. 5.8 into Eq. 5.24, the left side of the equation drops out:

$$\begin{aligned} L_{\text{CE}}(\hat{y}, y) &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y) \log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -[\log (1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \\ &= -\log (.30) \\ &= 1.2 \end{aligned}$$

Sure enough, the loss for the first classifier (.36) is less than the loss for the second classifier (1.2).

Why does minimizing this negative log probability do what we want? A perfect classifier would assign probability 1 to the correct outcome ($y = 1$ or $y = 0$) and probability 0 to the incorrect outcome. That means if y equals 1, the higher $\hat{y}$ is (the closer it is to 1), the better the classifier; the lower $\hat{y}$ is (the closer it is to 0), the worse the classifier. If y equals 0, instead, the higher $1 - \hat{y}$ is (closer to 1), the better the classifier. The negative log of $\hat{y}$ (if the true y equals 1) or $1 - \hat{y}$ (if the true y equals 0) is a convenient loss metric since it goes from 0 (negative log of 1, no loss) to infinity (negative log of 0, infinite loss). This loss function also ensures that as the probability of the correct answer is maximized, the probability of the incorrect answer is minimized; since the two sum to one, any increase in the probability of the correct answer is coming at the expense of the incorrect answer. It's called the cross-entropy loss, because Eq. 5.22 is also the formula for the **cross-entropy** between the true probability distribution y and our estimated distribution $\hat{y}$.

Now we know what we want to minimize; in the next section, we'll see how to find the minimum.

5.6 Gradient Descent

Our goal with gradient descent is to find the optimal weights: minimize the loss function we've defined for the model. In Eq. 5.25 below, we'll explicitly represent the fact that the cross-entropy loss function L_{CE} is parameterized by the weights. In

machine learning in general we refer to the parameters being learned as θ; in the case of logistic regression $\theta = \{\mathbf{w}, b\}$. So the goal is to find the set of weights which minimizes the loss function, averaged over all examples:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)}) \tag{5.25}$$

How shall we find the minimum of this (or any) loss function? Gradient descent is a method that finds a minimum of a function by figuring out in which direction (in the space of the parameters θ) the function's slope is rising the most steeply, and moving in the opposite direction. The intuition is that if you are hiking in a canyon and trying to descend most quickly down to the river at the bottom, you might look around yourself in all directions, find the direction where the ground is sloping the steepest, and walk downhill in that direction.

convex

For logistic regression, this loss function is conveniently **convex**. A convex function has at most one minimum; there are no local minima to get stuck in, so gradient descent starting from any point is guaranteed to find the minimum. (By contrast, the loss for multi-layer neural networks is non-convex, and gradient descent may get stuck in local minima for neural network training and never find the global optimum.)

Although the algorithm (and the concept of gradient) are designed for direction *vectors*, let's first consider a visualization of the case where the parameter of our system is just a single scalar w, shown in Fig. 5.4.

Given a random initialization of w at some value w^1, and assuming the loss function L happened to have the shape in Fig. 5.4, we need the algorithm to tell us whether at the next iteration we should move left (making w^2 smaller than w^1) or right (making w^2 bigger than w^1) to reach the minimum.

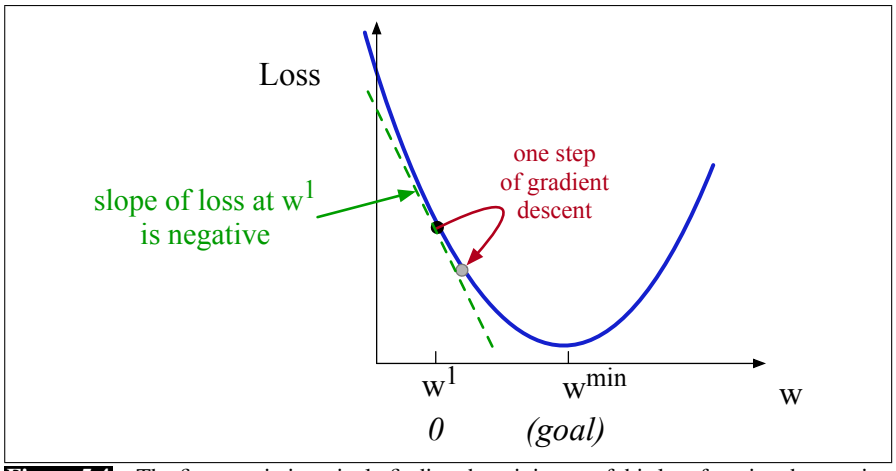

Figure 5.4 The first step in iteratively finding the minimum of this loss function, by moving w in the reverse direction from the slope of the function. Since the slope is negative, we need to move w in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of w (which is 0), w^2 the value at the second step, and so on.

gradient

The gradient descent algorithm answers this question by finding the **gradient** of the loss function at the current point and moving in the opposite direction. The gradient of a function of many variables is a vector pointing in the direction of the greatest increase in a function. The gradient is a multi-variable generalization of the

slope, so for a function of one variable like the one in Fig. 5.4, we can informally think of the gradient as the slope. The dotted line in Fig. 5.4 shows the slope of this hypothetical loss function at point $w = w^1$. You can see that the slope of this dotted line is negative. Thus to find the minimum, gradient descent tells us to go in the opposite direction: moving w in a positive direction.

learning rate

The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{dw}L(f(x;w),y)$ weighted by a **learning rate** η. A higher (faster) learning rate means that we should move w more on each step. The change we make in our parameter is the learning rate times the gradient (or the slope, in our single-variable example):

$$w^{t+1} = w^t - \eta \frac{d}{dw}L(f(x;w),y) \tag{5.26}$$

Now let's extend the intuition from a function of one scalar variable w to many variables, because we don't just want to move left or right, we want to know where in the N-dimensional space (of the N parameters that make up θ) we should move. The **gradient** is just such a vector; it expresses the directional components of the sharpest slope along each of those N dimensions. If we're just imagining two weight dimensions (say for one weight w and one bias b), the gradient might be a vector with two orthogonal components, each of which tells us how much the ground slopes in the w dimension and in the b dimension. Fig. 5.5 shows a visualization of the value of a 2-dimensional gradient vector taken at the red point.

In an actual logistic regression, the parameter vector $\mathbf{w}$ is much longer than 1 or 2, since the input feature vector $\mathbf{x}$ can be quite long, and we need a weight w_i for each x_i. For each dimension/variable w_i in $\mathbf{w}$ (plus the bias b), the gradient will have a component that tells us the slope with respect to that variable. In each dimension w_i, we express the slope as a partial derivative $\frac{\partial}{\partial w_i}$ of the loss function. Essentially we're asking: "How much would a small change in that variable w_i influence the total loss function L?"

Formally, then, the gradient of a multi-variable function f is a vector in which each component expresses the partial derivative of f with respect to one of the variables. We'll use the inverted Greek delta symbol ∇ to refer to the gradient, and

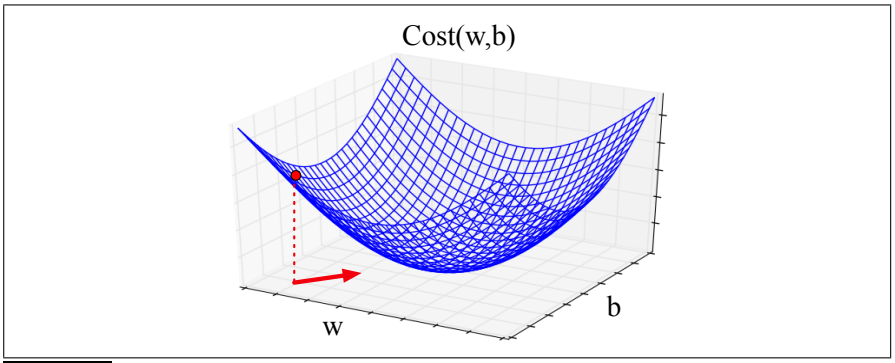

Figure 5.5 Visualization of the gradient vector at the red point in two dimensions w and b, showing a red arrow in the x-y plane pointing in the direction we will go to look for the minimum: the opposite direction of the gradient (recall that the gradient points in the direction of increase not decrease).

represent $\hat{y}$ as $f(x;\theta)$ to make the dependence on θ more obvious:

$$\nabla L(f(x;\theta),y) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \\ \frac{\partial}{\partial b} L(f(x;\theta),y) \end{bmatrix} \tag{5.27}$$

The final equation for updating θ based on the gradient is thus

$$\theta^{t+1} = \theta^t - \eta \nabla L(f(x;\theta),y) \tag{5.28}$$

5.6.1 The Gradient for Logistic Regression

In order to update θ, we need a definition for the gradient $\nabla L(f(x;\theta),y)$. Recall that for logistic regression, the cross-entropy loss function is:

$$L_{\text{CE}}(\hat{y},y) = -[y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \tag{5.29}$$

It turns out that the derivative of this function for one observation vector x is Eq. 5.30 (the interested reader can see Section 5.10 for the derivation of this equation):

$$\begin{aligned} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b) - y]x_j \\ &= (\hat{y}-y)x_j \end{aligned} \tag{5.30}$$

You'll also sometimes see this equation in the equivalent form:

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = -(y-\hat{y})x_j \tag{5.31}$$

Note in these equations that the gradient with respect to a single weight w_j represents a very intuitive value: the difference between the true y and our estimated $\hat{y} = \sigma(\mathbf{w}\cdot\mathbf{x}+b)$ for that observation, multiplied by the corresponding input value x_j.

5.6.2 The Stochastic Gradient Descent Algorithm

Stochastic gradient descent is an online algorithm that minimizes the loss function by computing its gradient after each training example, and nudging θ in the right direction (the opposite direction of the gradient). (An "online algorithm" is one that processes its input example by example, rather than waiting until it sees the entire input.) Stochastic gradient descent is called **stochastic** because it chooses a single random example at a time; in Section 5.6.4 we'll discuss other versions of gradient descent that batch many examples at once. Fig. 5.6 shows the algorithm.

hyperparameter

The learning rate η is a **hyperparameter** that must be adjusted. If it's too high, the learner will take steps that are too large, overshooting the minimum of the loss function. If it's too low, the learner will take steps that are too small, and take too long to get to the minimum. It is common to start with a higher learning rate and then slowly decrease it, so that it is a function of the iteration k of training; the notation η_k can be used to mean the value of the learning rate at iteration k.

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ
  # where: L is the loss function
  # f is a function parameterized by θ
  # x is the set of training inputs $x^{(1)}$, $x^{(2)}$,..., $x^{(m)}$
  # y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$,..., $y^{(m)}$

$\theta \leftarrow 0$ # (or small random values)
repeat til done # see caption
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
    1. Optional (for reporting): # How are we doing on this tuple?
      Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
      Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
    2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
    3. $\theta \leftarrow \theta - \eta\, g$ # Go the other way instead
return θ

Figure 5.6 The stochastic gradient descent algorithm. Step 1 (computing the loss) is used mainly to report how well we are doing on the current tuple; we don't need to compute the loss in order to compute the gradient. The algorithm can terminate when it converges (when the gradient norm $< \epsilon$), or when progress halts (for example when the loss starts going up on a held-out set). Weights are initialized to 0 for logistic regression, but to small random values for neural networks, as we'll see in Chapter 7.

We'll discuss hyperparameters in more detail in Chapter 7, but in short, they are a special kind of parameter for any machine learning model. Unlike regular parameters of a model (weights like w and b), which are learned by the algorithm from the training set, hyperparameters are special parameters chosen by the algorithm designer that affect how the algorithm works.

5.6.3 Working through an example

Let's walk through a single step of the gradient descent algorithm. We'll use a simplified version of the example in Fig. 5.2 as it sees a single observation x, whose correct value is $y = 1$ (this is a positive review), and with a feature vector $\mathbf{x} = [x_1, x_2]$ consisting of these two features:

$$\begin{aligned} x_1 &= 3 \quad \text{(count of positive lexicon words)} \\ x_2 &= 2 \quad \text{(count of negative lexicon words)} \end{aligned}$$

Let's assume the initial weights and bias in θ^0 are all set to 0, and the initial learning rate η is 0.1:

$$\begin{aligned} w_1 = w_2 = b &= 0 \\ \eta &= 0.1 \end{aligned}$$

The single update step requires that we compute the gradient, multiplied by the learning rate

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

In our mini example there are three parameters, so the gradient vector has 3 dimensions, for w_1, w_2, and b. We can compute the first gradient as follows:

$$\nabla_{w,b}L = \begin{bmatrix} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_1} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_2} \\ \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)x_1 \\ (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)x_2 \\ \sigma(\mathbf{w}\cdot\mathbf{x}+b)-y \end{bmatrix} = \begin{bmatrix} (\sigma(0)-1)x_1 \\ (\sigma(0)-1)x_2 \\ \sigma(0)-1 \end{bmatrix} = \begin{bmatrix} -0.5x_1 \\ -0.5x_2 \\ -0.5 \end{bmatrix} = \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix}$$

Now that we have a gradient, we compute the new parameter vector θ^1 by moving θ^0 in the opposite direction from the gradient:

$$\theta^1 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

So after one step of gradient descent, the weights have shifted to be: $w_1 = .15$, $w_2 = .1$, and $b = .05$.

Note that this observation x happened to be a positive example. We would expect that after seeing more negative examples with high counts of negative words, that the weight $\mathbf{w}_2$ would shift to have a negative value.

5.6.4 Mini-batch training

Stochastic gradient descent is called stochastic because it chooses a single random example at a time, moving the weights so as to improve performance on that single example. That can result in very choppy movements, so it's common to compute the gradient over batches of training instances rather than a single instance.

batch training

For example in **batch training** we compute the gradient over the entire dataset. By seeing so many examples, batch training offers a superb estimate of which direction to move the weights, at the cost of spending a lot of time processing every single example in the training set to compute this perfect direction.

mini-batch

A compromise is **mini-batch** training: we train on a group of m examples (perhaps 512, or 1024) that is less than the whole dataset. (If m is the size of the dataset, then we are doing **batch** gradient descent; if $m = 1$, we are back to doing stochastic gradient descent.) Mini-batch training also has the advantage of computational efficiency. The mini-batches can easily be vectorized, choosing the size of the mini-batch based on the computational resources. This allows us to process all the examples in one mini-batch in parallel and then accumulate the loss, something that's not possible with individual or batch training.

We just need to define mini-batch versions of the cross-entropy loss function we defined in Section 5.5 and the gradient in Section 5.6.1. Let's extend the cross-entropy loss for one example from Eq. 5.23 to mini-batches of size m. We'll continue to use the notation that $x^{(i)}$ and $y^{(i)}$ mean the ith training features and training label, respectively. We make the assumption that the training examples are independent:

$$\begin{aligned} \log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) \\ &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \end{aligned} \tag{5.32}$$

Now the cost function for the mini-batch of m examples is the average loss for each example:

$$\begin{aligned} Cost(\hat{y},y) &= \frac{1}{m}\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)},y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b)+(1-y^{(i)})\log\left(1-\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b)\right) \end{aligned} \tag{5.33}$$

The mini-batch gradient is the average of the individual gradients from Eq. 5.30:

$$\frac{\partial Cost(\hat{y},y)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left[\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b)-y^{(i)}\right]x_j^{(i)} \tag{5.34}$$

Instead of using the sum notation, we can more efficiently compute the gradient in its matrix form, following the vectorization we saw on page 7, where we have a matrix $\mathbf{X}$ of size $[m\times f]$ representing the m inputs in the batch, and a vector $\mathbf{y}$ of size $[m\times 1]$ representing the correct outputs:

$$\begin{aligned} \frac{\partial Cost(\hat{y},y)}{\partial \mathbf{w}} &= \frac{1}{m}(\hat{\mathbf{y}}-\mathbf{y})^{\intercal}\mathbf{X} \\ &= \frac{1}{m}(\sigma(\mathbf{X}\mathbf{w}+\mathbf{b})-\mathbf{y})^{\intercal}\mathbf{X} \end{aligned} \tag{5.35}$$

5.7 Regularization

Numquam ponenda est pluralitas sine necessitate
'Plurality should never be proposed unless needed'
William of Occam

There is a problem with learning weights that make the model perfectly match the training data. If a feature is perfectly predictive of the outcome because it happens to only occur in one class, it will be assigned a very high weight. The weights for features will attempt to perfectly fit details of the training set, in fact too perfectly, modeling noisy factors that just accidentally correlate with the class. This problem is called **overfitting**. A good model should be able to **generalize** well from the training data to the unseen test set, but a model that overfits will have poor generalization.

overfitting
generalize
regularization

To avoid overfitting, a new **regularization** term $R(\theta)$ is added to the loss function in Eq. 5.25, resulting in the following loss for a batch of m examples (slightly rewritten from Eq. 5.25 to be maximizing log probability rather than minimizing loss, and removing the $\frac{1}{m}$ term which doesn't affect the argmax):

$$\hat{\theta} = \operatorname*{argmax}_{\theta}\sum_{i=1}^{m}\log P(y^{(i)}|x^{(i)}) - \alpha R(\theta) \tag{5.36}$$

The new regularization term $R(\theta)$ is used to penalize large weights. Thus a setting of the weights that matches the training data perfectly— but uses many weights with

high values to do so—will be penalized more than a setting that matches the data a little less well, but does so using smaller weights. There are two common ways to compute this regularization term $R(\theta)$. **L2 regularization** is a quadratic function of the weight values, named because it uses the (square of the) L2 norm of the weight values. The L2 norm, $||\theta||_2$, is the same as the **Euclidean distance** of the vector θ from the origin. If θ consists of n weights, then:

L2 regularization

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2 \tag{5.37}$$

The L2 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} \theta_j^2 \tag{5.38}$$

L1 regularization

L1 regularization is a linear function of the weight values, named after the L1 norm $||W||_1$, the sum of the absolute values of the weights, or **Manhattan distance** (the Manhattan distance is the distance you'd have to walk between two points in a city with a street grid like New York):

$$R(\theta) = ||\theta||_1 = \sum_{i=1}^{n} |\theta_i| \tag{5.39}$$

The L1 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |\theta_j| \tag{5.40}$$

lasso
ridge

These kinds of regularization come from statistics, where L1 regularization is called **lasso regression** (Tibshirani, 1996) and L2 regularization is called **ridge regression**, and both are commonly used in language processing. L2 regularization is easier to optimize because of its simple derivative (the derivative of θ^2 is just 2θ), while L1 regularization is more complex (the derivative of $|\theta|$ is non-continuous at zero). But while L2 prefers weight vectors with many small weights, L1 prefers sparse solutions with some larger weights but many more weights set to zero. Thus L1 regularization leads to much sparser weight vectors, that is, far fewer features.

Both L1 and L2 regularization have Bayesian interpretations as constraints on the prior of how weights should look. L1 regularization can be viewed as a Laplace prior on the weights. L2 regularization corresponds to assuming that weights are distributed according to a Gaussian distribution with mean $\mu = 0$. In a Gaussian or normal distribution, the further away a value is from the mean, the lower its probability (scaled by the variance σ). By using a Gaussian prior on the weights, we are saying that weights prefer to have the value 0. A Gaussian for a weight θ_j is

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2} \right) \tag{5.41}$$

If we multiply each weight by a Gaussian prior on the weight, we are thus maximizing the following constraint:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) \times \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2} \right) \tag{5.42}$$

which in log space, with $\mu = 0$, and assuming $2\sigma^2 = 1$, corresponds to

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha \sum_{j=1}^{n} \theta_j^2 \tag{5.43}$$

which is in the same form as Eq. 5.38.

5.8 Learning in Multinomial Logistic Regression

The loss function for multinomial logistic regression generalizes the loss function for binary logistic regression from 2 to K classes. Recall that that the cross-entropy loss for binary logistic regression (repeated from Eq. 5.23) is:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{5.44}$$

The loss function for multinomial logistic regression generalizes the two terms in Eq. 5.44 (one that is non-zero when $y = 1$ and one that is non-zero when $y = 0$) to K terms. As we mentioned above, for multinomial regression we'll represent both $\mathbf{y}$ and $\hat{\mathbf{y}}$ as vectors. The true label $\mathbf{y}$ is a vector with K elements, each corresponding to a class, with $y_c = 1$ if the correct class is c, with all other elements of $\mathbf{y}$ being 0. And our classifier will produce an estimate vector with K elements $\hat{\mathbf{y}}$, each element $\hat{y}_k$ of which represents the estimated probability $p(y_k = 1|\mathbf{x})$.

The loss function for a single example $\mathbf{x}$, generalizing from binary logistic regression, is the sum of the logs of the K output classes, each weighted by the indicator function $\mathbf{y}_k$ (Eq. 5.45). This turns out to be just the negative log probability of the correct class c (Eq. 5.46):

$$\begin{aligned} L_{\text{CE}}(\hat{\mathbf{y}}, \mathbf{y}) &= -\sum_{k=1}^{K} y_k \log \hat{y}_k && (5.45) \\ &= -\log \hat{y}_c, \quad \text{(where } c \text{ is the correct class)} && (5.46) \\ &= -\log \hat{p}(y_c = 1|\mathbf{x}) \quad \text{(where } c \text{ is the correct class)} \\ &= -\log \frac{\exp(\mathbf{w_c} \cdot \mathbf{x} + b_c)}{\sum_{j=1}^{K} \exp(\mathbf{w_j} \cdot \mathbf{x} + b_j)} \quad (c \text{ is the correct class}) && (5.47) \end{aligned}$$

How did we get from Eq. 5.45 to Eq. 5.46? Because only one class (let's call it c) is the correct one, the vector $\mathbf{y}$ takes the value 1 only for this value of k, i.e., has $y_c = 1$ and $y_j = 0 \ \ \forall j \neq c$. That means the terms in the sum in Eq. 5.45 will all be 0 except for the term corresponding to the true class c. Hence the cross-entropy loss is simply the log of the output probability corresponding to the correct class, and we therefore also call Eq. 5.46 the **negative log likelihood loss**.

negative log likelihood loss

Of course for gradient descent we don't need the loss, we need its gradient. The gradient for a single example turns out to be very similar to the gradient for binary logistic regression, $(\hat{y} - y)x$, that we saw in Eq. 5.30. Let's consider one piece of the gradient, the derivative for a single weight. For each class k, the weight of the ith element of input $\mathbf{x}$ is $w_{k,i}$. What is the partial derivative of the loss with respect to $w_{k,i}$? This derivative turns out to be just the difference between the true value for the class k (which is either 1 or 0) and the probability the classifier outputs for class k,

weighted by the value of the input x_i corresponding to the ith element of the weight vector for class k:

$$\begin{aligned}\frac{\partial L_{\text{CE}}}{\partial w_{k,i}} &= -(y_k - \hat{y}_k)x_i \\ &= -(y_k - p(y_k = 1|x))x_i \\ &= -\left(y_k - \frac{\exp(\mathbf{w_k}\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)}\right)x_i \end{aligned} \tag{5.48}$$

We'll return to this case of the gradient for softmax regression when we introduce neural networks in Chapter 7, and at that time we'll also discuss the derivation of this gradient in equations Eq. **??**–Eq. **??**.

5.9 Interpreting models

interpretable

Often we want to know more than just the correct classification of an observation. We want to know why the classifier made the decision it did. That is, we want our decision to be **interpretable**. Interpretability can be hard to define strictly, but the core idea is that as humans we should know why our algorithms reach the conclusions they do. Because the features to logistic regression are often human-designed, one way to understand a classifier's decision is to understand the role each feature plays in the decision. Logistic regression can be combined with statistical tests (the likelihood ratio test, or the Wald test); investigating whether a particular feature is significant by one of these tests, or inspecting its magnitude (how large is the weight w associated with the feature?) can help us interpret why the classifier made the decision it makes. This is enormously important for building transparent models.

Furthermore, in addition to its use as a classifier, logistic regression in NLP and many other fields is widely used as an analytic tool for testing hypotheses about the effect of various explanatory variables (features). In text classification, perhaps we want to know if logically negative words (*no, not, never*) are more likely to be associated with negative sentiment, or if negative reviews of movies are more likely to discuss the cinematography. However, in doing so it's necessary to control for potential confounds: other factors that might influence sentiment (the movie genre, the year it was made, perhaps the length of the review in words). Or we might be studying the relationship between NLP-extracted linguistic features and non-linguistic outcomes (hospital readmissions, political outcomes, or product sales), but need to control for confounds (the age of the patient, the county of voting, the brand of the product). In such cases, logistic regression allows us to test whether some feature is associated with some outcome above and beyond the effect of other features.

5.10 Advanced: Deriving the Gradient Equation

In this section we give the derivation of the gradient of the cross-entropy loss function L_{CE} for logistic regression. Let's start with some quick calculus refreshers. First, the derivative of $\ln(x)$:

$$\frac{d}{dx}\ln(x) = \frac{1}{x} \tag{5.49}$$

Second, the (very elegant) derivative of the sigmoid:

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{5.50}$$

chain rule

Finally, the **chain rule** of derivatives. Suppose we are computing the derivative of a composite function $f(x) = u(v(x))$. The derivative of $f(x)$ is the derivative of $u(x)$ with respect to $v(x)$ times the derivative of $v(x)$ with respect to x:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{5.51}$$

First, we want to know the derivative of the loss function with respect to a single weight w_j (we'll need to compute it for each weight, and for the bias):

$$\begin{aligned}\frac{\partial L_{\text{CE}}}{\partial w_j} &= \frac{\partial}{\partial w_j} -[y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -\left[\frac{\partial}{\partial w_j} y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + \frac{\partial}{\partial w_j}(1-y)\log[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\right]\end{aligned} \tag{5.52}$$

Next, using the chain rule, and relying on the derivative of log:

$$\frac{\partial L_{\text{CE}}}{\partial w_j} = -\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b) - \frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \tag{5.53}$$

Rearranging terms:

$$\frac{\partial L_{\text{CE}}}{\partial w_j} = -\left[\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)} - \frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\right]\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b) \tag{5.54}$$

And now plugging in the derivative of the sigmoid, and using the chain rule one more time, we end up with Eq. 5.55:

$$\begin{aligned}\frac{\partial L_{\text{CE}}}{\partial w_j} &= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\frac{\partial(\mathbf{w}\cdot\mathbf{x}+b)}{\partial w_j} \\ &= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j \\ &= -[y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j \\ &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j\end{aligned} \tag{5.55}$$

5.11 Summary

This chapter introduced the **logistic regression** model of **classification**.

- Logistic regression is a supervised machine learning classifier that extracts real-valued features from the input, multiplies each by a weight, sums them, and passes the sum through a **sigmoid** function to generate a probability. A threshold is used to make a decision.

- Logistic regression can be used with two classes (e.g., positive and negative sentiment) or with multiple classes (**multinomial logistic regression**, for example for n-ary text classification, part-of-speech labeling, etc.).
- Multinomial logistic regression uses the **softmax** function to compute probabilities.
- The weights (vector w and bias b) are learned from a labeled training set via a loss function, such as the **cross-entropy loss**, that must be minimized.
- Minimizing this loss function is a **convex optimization** problem, and iterative algorithms like **gradient descent** are used to find the optimal weights.
- **Regularization** is used to avoid overfitting.
- Logistic regression is also one of the most useful analytic tools, because of its ability to transparently study the importance of individual features.

Bibliographical and Historical Notes

Logistic regression was developed in the field of statistics, where it was used for the analysis of binary data by the 1960s, and was particularly common in medicine (Cox, 1969). Starting in the late 1970s it became widely used in linguistics as one of the formal foundations of the study of linguistic variation (Sankoff and Labov, 1979).

Nonetheless, logistic regression didn't become common in natural language processing until the 1990s, when it seems to have appeared simultaneously from two directions. The first source was the neighboring fields of information retrieval and speech processing, both of which had made use of regression, and both of which lent many other statistical techniques to NLP. Indeed a very early use of logistic regression for document routing was one of the first NLP applications to use (LSI) embeddings as word representations (Schütze et al., 1995).

maximum entropy

At the same time in the early 1990s logistic regression was developed and applied to NLP at IBM Research under the name **maximum entropy** modeling or **maxent** (Berger et al., 1996), seemingly independent of the statistical literature. Under that name it was applied to language modeling (Rosenfeld, 1996), part-of-speech tagging (Ratnaparkhi, 1996), parsing (Ratnaparkhi, 1997), coreference resolution (Kehler, 1997), and text classification (Nigam et al., 1999).

More on classification can be found in machine learning textbooks (Hastie et al. 2001, Witten and Frank 2005, Bishop 2006, Murphy 2012).

Exercises

Berger, A., S. A. Della Pietra, and V. J. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Bishop, C. M. 2006. *Pattern recognition and machine learning*. Springer.

Cox, D. 1969. *Analysis of Binary Data*. Chapman and Hall, London.

Hastie, T., R. J. Tibshirani, and J. H. Friedman. 2001. *The Elements of Statistical Learning*. Springer.

Kehler, A. 1997. Probabilistic coreference in information extraction. *EMNLP*.

Murphy, K. P. 2012. *Machine learning: A probabilistic perspective*. MIT Press.

Ng, A. Y. and M. I. Jordan. 2002. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *NeurIPS*.

Nigam, K., J. D. Lafferty, and A. McCallum. 1999. Using maximum entropy for text classification. *IJCAI-99 workshop on machine learning for information filtering*.

Ratnaparkhi, A. 1996. A maximum entropy part-of-speech tagger. *EMNLP*.

Ratnaparkhi, A. 1997. A linear observed time statistical parser based on maximum entropy models. *EMNLP*.

Rosenfeld, R. 1996. A maximum entropy approach to adaptive statistical language modeling. *Computer Speech and Language*, 10:187–228.

Sankoff, D. and W. Labov. 1979. On the uses of variable rules. *Language in society*, 8(2-3):189–222.

Schütze, H., D. A. Hull, and J. Pedersen. 1995. A comparison of classifiers and document representations for the routing problem. *SIGIR-95*.

Tibshirani, R. J. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288.

Wang, S. and C. D. Manning. 2012. Baselines and bigrams: Simple, good sentiment and topic classification. *ACL*.

Witten, I. H. and E. Frank. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd edition. Morgan Kaufmann.