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 cues and some particular outcome: **logistic regression**. Indeed, logistic regression is one of the most important analytic tool 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 5.6.

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.

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

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

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 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 observations 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 for learning, usually involving minimizing 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) 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 Classification: the sigmoid

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, and $P(y=1|x)$ is the probability that the document has positive sentiment, while 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 (meaning the feature is associated with the class) or negative (meaning the feature is not associated with the 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 a and b, written as $a \cdot b$ is the sum of the products of the corresponding elements of each vector. Thus the following is an equivalent formation to Eq. 5.2:

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

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 ∞.

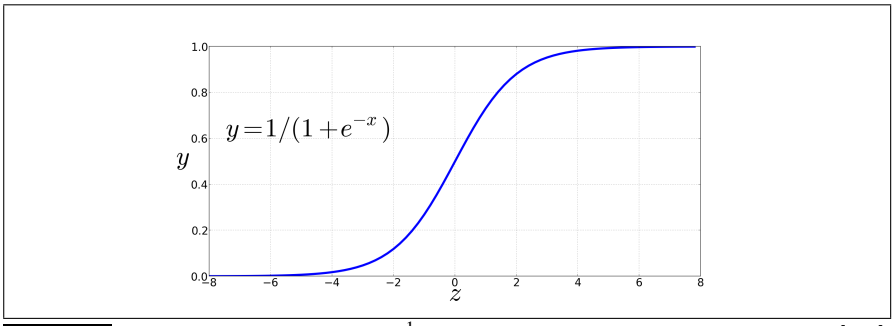

Figure 5.1 The sigmoid function $y = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $[0,1]$. Because it is nearly linear around 0 but has a sharp slope toward the ends, it tends to squash outlier values toward 0 or 1.

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:

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

The sigmoid has a number of advantages; it take 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 has a sharp slope toward the ends, it tends to squash outlier values toward 0 or 1. And it's differentiable, which as we'll see in Section 5.8 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:

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

decision boundary

Now we have an algorithm that given an instance x computes the probability $P(y=1|x)$. How do we make a decision? For a test instance x, we say yes if the probability $P(y=1|x)$ is more than .5, and no otherwise. We call .5 the **decision boundary**:

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

5.1.1 Example: sentiment classification

Let's have an example. 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 following 6 features $x_1 ... x_6$ of the input; Fig. 5.2 shows the features in a sample mini test document.

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ 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 $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(64) = 4.15$

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

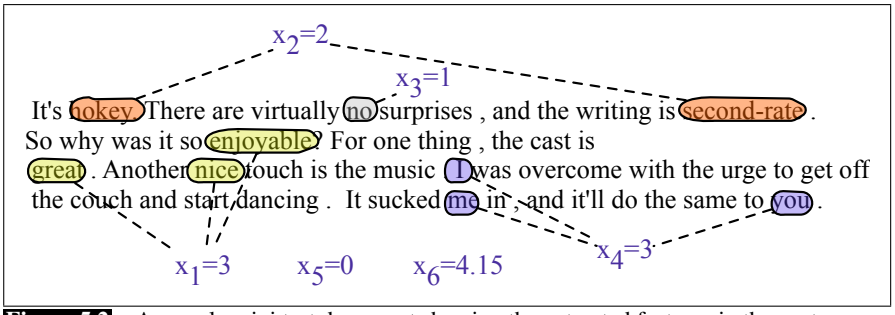

Figure 5.2 A sample mini test document showing the extracted features in the vector x.

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:

$$\begin{aligned} p(+|x) = P(Y=1|x) &= \sigma(w \cdot x + b) \\ &= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.15] + 0.1) \\ &= \sigma(1.805) \\ &= 0.86 \\ p(-|x) = P(Y=0|x) &= 1 - \sigma(w \cdot x + b) \\ &= 0.14 \end{aligned}$$

Logistic regression is commonly 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 and the class is EOS (perhaps with a positive weight), or that the current word is in our abbreviations dictionary ("Prof.") and the class is EOS (perhaps with a negative weight). A feature can also express a quite complex combination of properties. For example a period following a upper cased word is a 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*.

$$\begin{aligned} x_1 &= \begin{cases} 1 & \text{if } \text{“}Case(w_i) = \text{Lower”} \\ 0 & \text{otherwise} \end{cases} \\ x_2 &= \begin{cases} 1 & \text{if } \text{“}w_i \in \text{AcronymDict”} \\ 0 & \text{otherwise} \end{cases} \\ x_3 &= \begin{cases} 1 & \text{if } \text{“}w_i = \text{St. \& } Case(w_{i-1}) = \text{Cap”} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Designing features: Features are generally designed by examining the training set with an eye to linguistic intuitions and the linguistic literature on the domain. A careful error analysis on the training or dev set. of an early version of a system often provides insights into features.

For some tasks it is especially helpful to build 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 sentence if the previous word was capitalized. For logistic regression and naive Bayes these combination features or **feature interactions** have to be designed by hand.

feature interactions

For many tasks (especially when feature values can reference specific words) we'll need large numbers of features. Often these are 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 templates

In order to avoid the extensive human effort of feature design, recent research in NLP has focused 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.

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 works extremely well (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.

5.2 Learning in Logistic Regression

How are the parameters of the model, the weights 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 w and b) that make $\hat{y}$ for each training observation as close as possible to the true y .

This requires 2 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

loss

function that is commonly used for logistic regression and also for neural networks, the **cross-entropy 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.

5.3 The cross-entropy loss function

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

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y \tag{5.6}$$

You could imagine using a simple loss function that just takes the mean squared error between $\hat{y}$ and y.

$$L_{\text{MSE}}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2 \tag{5.7}$$

It turns out that this MSE loss, which is very useful for some algorithms like linear regression, becomes harder to optimize (technically, non-convex), when it's applied to probabilistic classification.

Instead, we use a loss function that prefers the correct class labels of the training example 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. 5.8 simplifies to $\hat{y}$; if y=0, Eq. 5.8 simplifies to $1 - \hat{y}$):

$$p(y|x) = \hat{y}^y (1 - \hat{y})^{1-y} \tag{5.8}$$

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:

$$\begin{aligned} \log p(y|x) &= \log \left[\hat{y}^y (1 - \hat{y})^{1-y}\right] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \end{aligned} \tag{5.9}$$

Eq. 5.9 describes a log likelihood that should be maximized. In order to turn this into loss function (something that we need to minimize), we'll just flip the sign on Eq. 5.9. The result is the cross-entropy loss L_{CE}:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \tag{5.10}$$

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

$$L_{CE}(w, b) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log (1 - \sigma(w \cdot x + b))] \tag{5.11}$$

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 the higher $\hat{y}$ (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. The negative log of this probability 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 insures that as 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. 5.9 is also the formula for the **cross-entropy** between the true probability distribution y and our estimated distribution $\hat{y}$.

Let's now extend Eq. 5.10 from one example to the whole training set: 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:

$$\begin{aligned}\log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) && (5.12)\\ &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) && (5.13)\\ &= -\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)}) && (5.14)\end{aligned}$$

We'll define the cost function for the whole dataset as the average loss for each example:

$$\begin{aligned}Cost(w,b) &= \frac{1}{m}\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)})\\ &= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(w\cdot x^{(i)}+b) + (1-y^{(i)}) \log\left(1-\sigma(w\cdot x^{(i)}+b)\right)\end{aligned} \quad (5.15)$$

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

5.4 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. 5.16 below, we'll explicitly represent the fact that the loss function L is parameterized by the weights, which we'll refer to in machine learning in general as θ (in the case of logistic regression $\theta = w, b$):

$$\hat{\theta} = \underset{\theta}{\text{argmin}} \frac{1}{m}\sum_{i=1}^{m} L_{CE}(y^{(i)}, x^{(i)}; \theta) \quad (5.16)$$

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 360 degrees, 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 just one minimum; there are no local minima to get stuck in, so gradient descent starting from any point is guaranteed to find the minimum.

Although the algorithm (and the concept of gradient) are designed for direction *vectors*, let's first consider a visualization of the the case where the parameter of our system, is just a single scalar w, shown in Fig. 5.3.

Given a random initialization of w at some value w_1, and assuming the loss function L happened to have the shape in Fig. 5.3, 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.

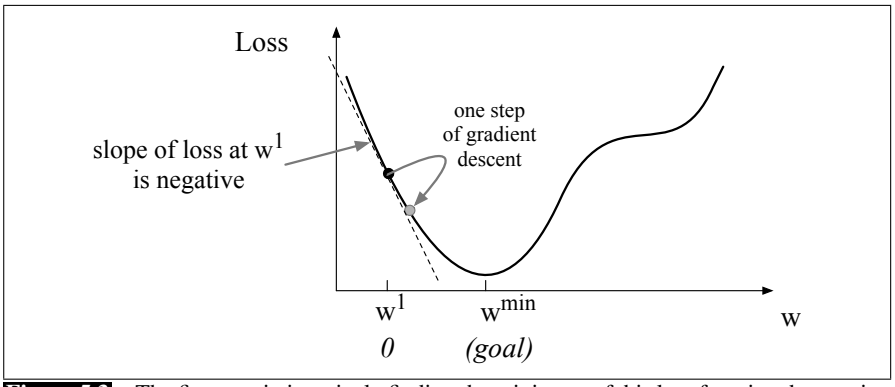

Figure 5.3 The first step in iteratively finding the minimum of this loss function, by moving w in the reverse direction from the slope of the function. Since the slope is negative, we need to move w in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of w (which is 0), w^2 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 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. 5.3, we can informally think of the gradient as the slope. The dotted line in Fig. 5.3 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 $\frac{d}{dw}f(x;w)$ 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):

$$w^{t+1} = w^t - \eta \frac{d}{dw} f(x;w) \tag{5.17}$$

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 dimension (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. 5.4 shows a visualization:

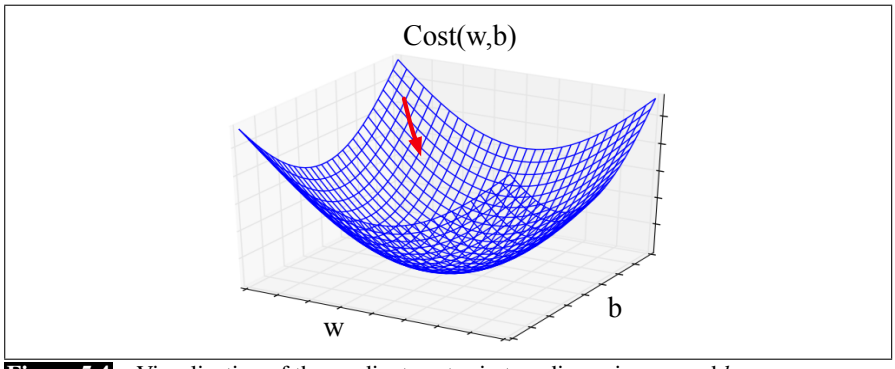

Figure 5.4 Visualization of the gradient vector in two dimensions w and b.

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

In each dimension w_i, we express the slope as a partial derivative $\frac{\partial}{\partial w_i}$ of the loss function. The gradient is then defined as a vector of these partials. We'll represent $\hat{y}$ as $f(x;\theta)$ to make the dependence on θ more obvious:

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \end{bmatrix} \tag{5.18}$$

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

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y) \tag{5.19}$$

5.4.1 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:

$$L_{CE}(w,b) = -[y \log \sigma(w \cdot x + b) + (1-y) \log (1 - \sigma(w \cdot x + b))] \tag{5.20}$$

It turns out that the derivative of this function for one observation vector x is Eq. 5.21 (the interested reader can see Section 5.8 for the derivation of this equation):

$$\frac{\partial L_{CE}(w,b)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j \tag{5.21}$$

Note in Eq. 5.21 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 $\hat{y} = \sigma(w \cdot x + b)$ for that observation, multiplied by the corresponding input value x_j.

The loss for a batch of data or an entire dataset is just the average loss over the m examples:

$$Cost(w,b) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(w \cdot x^{(i)} + b) + (1 - y^{(i)}) \log \left(1 - \sigma(w \cdot x^{(i)} + b)\right) \tag{5.22}$$

And the gradient for multiple data points is the sum of the individual gradients::

$$\frac{\partial Cost(w,b)}{\partial w_j} = \sum_{i=1}^{m} \left[\sigma(w \cdot x^{(i)} + b) - y^{(i)}\right] x_j^{(i)} \tag{5.23}$$

5.4.2 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). Fig. 5.5 shows the algorithm.

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 $x^{(1)}$, $x^{(2)}$,..., $x^{(n)}$
  # y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$,..., $y^{(n)}$

$\theta \leftarrow 0$
repeat T times
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
  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 $y^{(i)}$?
  $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss ?
  $\theta \leftarrow \theta - \eta\, g$ # go the other way instead
return θ

Figure 5.5 The stochastic gradient descent algorithm

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 also common to do **minibatch** gradient descent, which computes the gradient over batches of training instances rather than a single instance.

minibatch

The learning rate η is a parameter 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 most common to begin the learning rate at a higher value, and then slowly decrease it, so that it is a function of the iteration k of training; you will sometimes see the notation η_k to mean the value of the learning rate at iteration k.

5.4.3 Working through an example

Let's walk though 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 only 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:

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

Now that we have a gradient, we compute the new parameter vector θ^2 by moving θ^1 in the opposite direction from the gradient:

$$\theta^2 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

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 w_2 would shift to have a negative value.

5.5 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 **regularization** term is added to the objective function in Eq. 5.16, resulting in the following objective:

$$\hat{w} = \underset{w}{\text{argmax}} \sum_{1=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(w) \tag{5.24}$$

The new component, $R(w)$ is called a regularization term, and 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.

L2 regularization

There are two common regularization terms $R(w)$. **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, $||W||_2$, is the same as the **Euclidean distance**:

$$R(W) = ||W||_2^2 = \sum_{j=1}^{N} w_j^2 \tag{5.25}$$

The L2 regularized objective function becomes:

$$\hat{w} = \underset{w}{\text{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} w_j^2 \tag{5.26}$$

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):

$$R(W) = ||W||_1 = \sum_{i=1}^{N} |w_i| \tag{5.27}$$

The L1 regularized objective function becomes:

$$\hat{w} = \underset{w}{\text{argmax}} \left[\sum_{1=i}^{m} \log P(y^{(i)}|x^{(i)}) \right] - \alpha \sum_{j=1}^{n} |w_j| \tag{5.28}$$

These kinds of regularization come from statistics, where L1 regularization is called the **'lasso'** or **lasso regression** (Tibshirani, 1996) and L2 regression 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 w^2 is just $2w$), while L1 regularization is more complex (the derivative of $|w|$ is non-continuous at zero). But where 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 w_j is

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(w_j-\mu_j)^2}{2\sigma_j^2}\right) \tag{5.29}$$

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

$$\hat{w} = \operatorname*{argmax}_w \prod_{i=1}^{M} P(y^{(i)}|x^{(i)}) \times \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(w_j-\mu_j)^2}{2\sigma_j^2}\right) \tag{5.30}$$

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

$$\hat{w} = \operatorname*{argmax}_w \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha \sum_{j=1}^{n} w_j^2 \tag{5.31}$$

which is in the same form as Eq. 5.26.

5.6 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 classifying the part of speech of a word (choosing from 10, 30, or even 50 different parts of speech), or assigning semantic labels like the named entities or semantic relations we will introduce in Chapter 17.

multinomial logistic regression

In such cases we use **multinominal logistic regression**, also called **softmax regression** (or, historically, the **maxent classifier**). In multinominal logistic regression the target y is a variable that ranges over more than two classes; we want to know the probability of y being in each potential class $c \in C$, $p(y = c|x)$.

softmax

The multinominal logistic classifier uses a generalization of the sigmoid, called the **softmax** function, to compute the probability $p(y = c|x)$. The softmax function takes a vector $z = [z_1, z_2, ..., z_k]$ 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 z of dimensionality k, the softmax is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \leq i \leq k \tag{5.32}$$

The softmax of an input vector $z = [z_1, z_2, ..., z_k]$ is thus a vector itself:

$$\text{softmax}(z) = \left[\frac{e^{z_1}}{\sum_{i=1}^{k} e^{z_i}}, \frac{e^{z_2}}{\sum_{i=1}^{k} e^{z_i}}, ..., \frac{e^{z_k}}{\sum_{i=1}^{k} e^{z_i}} \right] \tag{5.33}$$

The denominator $\sum_{i=1}^{k} e^{z_i}$ is used to normalize all the values into probabilities. Thus for example given a vector:

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

the result softmax(z) is

$$[0.055, 0.090, 0.0067, 0.10, 0.74, 0.010]$$

Again like the sigmoid, the input to the softmax will be the dot product between a weight vector w and an input vector x (plus a bias). But now we'll need separate weight vectors (and bias) for each of the K classes.

$$p(y = c|x) = \frac{e^{w_c \cdot x + b_c}}{\sum_{j=1}^{k} e^{w_j \cdot x + b_j}} \tag{5.34}$$

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, will tend to push its probability toward 1, and suppress the probabilities of the smaller inputs.

5.6.1 Features in Multinomial Logistic Regression

For multiclass classification the input features need to be a function of both the observation x and the candidate output class c. Thus instead of the notation x_i, f_i or $f_i(x)$, when we're discussing features we will use the notation $f_i(c,x)$, meaning feature i for a particular class c for a given observation x.

In binary classification, a positive weight on a feature pointed toward y=1 and a negative weight toward y=0... but in multiclass a feature could be evidence for or against an individual class.

Let's look at some sample features for a few NLP tasks to help understand this perhaps unintuitive use of features that are functions of both the observation x and the class c,

Suppose we are doing text classification, and instead of binary classification our task is to assign one of the 3 classes $+$, $-$, or 0 (neutral) to a document. Now a feature related to exclamation marks might have a negative weight for 0 documents, and a positive weight for $+$ or $-$ documents:

Var	Definition	Wt
$f_1(0,x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	-4.5
$f_1(+,x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	2.6
$f_1(0,x)$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1.3

5.6.2 Learning in Multinomial Logistic Regression

Multinomial logistic regression has a slightly different loss function than binary logistic regression because it uses the softmax rather than sigmoid classifier, The loss function for a single example x is the sum of the logs of the K output classes:

$$\begin{aligned} L_{CE}(\hat{y},y) &= -\sum_{k=1}^{K} 1\{y=k\} \log p(y=k|x) \\ &= -\sum_{k=1}^{K} 1\{y=k\} \log \frac{e^{w_k \cdot x + b_k}}{\sum_{j=1}^{K} e^{w_j \cdot x + b_j}} \end{aligned} \tag{5.35}$$

This makes use of the function $1\{\}$ which evaluates to 1 if the condition in the brackets is true and to 0 otherwise.

The gradient for a single example turns out to be very similar to the gradient for logistic regression, although we don't show the derivation here. It is the different between the value for the true class k (which is 1) and the probability the classifier outputs for class k, weighted by the value of the input x_k:

$$\begin{aligned} \frac{\partial L_{CE}}{\partial w_k} &= (1\{y=k\} - p(y=k|x))x_k \\ &= \left(1\{y=k\} - \frac{e^{w_k \cdot x + b_k}}{\sum_{j=1}^{K} e^{w_j \cdot x + b_j}}\right) x_k \end{aligned} \tag{5.36}$$

5.7 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 it 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.

5.8 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 $\ln(x)$:

$$\frac{d}{dx}\ln(x) = \frac{1}{x} \tag{5.37}$$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{5.38}$$

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:

$$\frac{df}{dx} = \frac{du}{dv}\cdot\frac{dv}{dx} \tag{5.39}$$

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):

$$\begin{aligned}\frac{\partial LL(w,b)}{\partial w_j} &= \frac{\partial}{\partial w_j} - [y\log\sigma(w\cdot x+b) + (1-y)\log(1-\sigma(w\cdot x+b))] \\ &= -\left[\frac{\partial}{\partial w_j} y\log\sigma(w\cdot x+b) + \frac{\partial}{\partial w_j}(1-y)\log[1-\sigma(w\cdot x+b)]\right]\end{aligned} \tag{5.40}$$

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

$$\frac{\partial LL(w,b)}{\partial w_j} = -\frac{y}{\sigma(w\cdot x+b)}\frac{\partial}{\partial w_j}\sigma(w\cdot x+b) - \frac{1-y}{1-\sigma(w\cdot x+b)}\frac{\partial}{\partial w_j}1-\sigma(w\cdot x+b) \tag{5.41}$$

Rearranging terms:

$$\frac{\partial LL(w,b)}{\partial w_j} = -\left[\frac{y}{\sigma(w\cdot x+b)} - \frac{1-y}{1-\sigma(w\cdot x+b)}\right]\frac{\partial}{\partial w_j}\sigma(w\cdot x+b) \tag{5.42}$$

And now plugging in the derivative of the sigmoid, and using the chain rule one more time, we end up with Eq. 5.43:

$$\begin{aligned}\frac{\partial LL(w,b)}{\partial w_j} &= -\left[\frac{y-\sigma(w\cdot x+b)}{\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]}\right]\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]\frac{\partial(w\cdot x+b)}{\partial w_j}\\ &= -\left[\frac{y-\sigma(w\cdot x+b)}{\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]}\right]\sigma(w\cdot x+b)[1-\sigma(w\cdot x+b)]x_j\\ &= -[y-\sigma(w\cdot x+b)]x_j\\ &= [\sigma(w\cdot x+b)-y]x_j \qquad (5.43)\end{aligned}$$

5.9 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), 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., Della Pietra, S. A., and Della Pietra, V. J. (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., Tibshirani, R. J., and Friedman, J. H. (2001). *The Elements of Statistical Learning*. Springer.

Murphy, K. P. (2012). *Machine learning: A probabilistic perspective*. MIT press.

Ng, A. Y. and Jordan, M. I. (2002). On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *NIPS 14*, pp. 841–848.

Nigam, K., Lafferty, J. D., and McCallum, A. (1999). Using maximum entropy for text classification. In *IJCAI-99 workshop on machine learning for information filtering*, pp. 61–67.

Ratnaparkhi, A. (1996). A maximum entropy part-of-speech tagger. In *EMNLP 1996*, Philadelphia, PA, pp. 133–142.

Ratnaparkhi, A. (1997). A linear observed time statistical parser based on maximum entropy models. In *EMNLP 1997*, Providence, RI, pp. 1–10.

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

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

Schütze, H., Hull, D. A., and Pedersen, J. (1995). A comparison of classifiers and document representations for the routing problem. In *SIGIR-95*, pp. 229–237.

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 Manning, C. D. (2012). Baselines and bigrams: Simple, good sentiment and topic classification. In *ACL 2012*, pp. 90–94.

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