CHAPTER 4 Logistic Regression and Text Classification

En sus remotas páginas está escrito que los animales se dividen en:

a. pertenecientes al Emperador	h. incluidos en esta clasificación
b. embalsamados	i. que se agitan como locos
c. amaestrados	j. innumerables
d. lechones	k. dibujados con un pincel finísimo de pelo de camello
e. sirenas	l. etcétera
f. fabulosos	m. que acaban de romper el jarrón
g. perros sueltos	n. que de lejos parecen moscas

Borges (1964)

Classification lies at the heart of language processing and intelligence. Recognizing a letter, a word, or a face, sorting mail, assigning grades to homeworks; these are all examples of assigning a category to an input. The challenges of classification were famously highlighted by the fabulist Jorge Luis Borges (1964), who imagined an ancient mythical encyclopedia that classified animals into:

(a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel's hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance.

text categorization

sentiment analysis

Luckily, the classes we use for language processing are easier to define than those of Borges. In this chapter we introduce the **logistic regression** algorithm for classification, and apply it to **text categorization**, the task of assigning a label or category to a text or document. We'll focus on one text categorization task, **sentiment analysis**, the categorization of **sentiment**, the positive or negative orientation that a writer expresses toward some object. A review of a movie, book, or product expresses the author's sentiment toward the product, while an editorial or political text expresses sentiment toward an action or candidate. Extracting sentiment is thus relevant for fields from marketing to politics.

For the binary task of labeling a text as indicating positive or negative stance, words (like *awesome* and *love*, or *awful* and *ridiculously* are very informative, as we can see from these sample extracts from movie/restaurant reviews:

$+$ *...awesome caramel sauce and sweet toasty almonds. I love this place!*
$-$ *...awful pizza and ridiculously overpriced...*

spam detection

language id

authorship attribution

There are many text classification tasks. In **spam detection** we assign an email to one of the two classes *spam* or *not-spam*. **Language id** is the task of determining what language a text is written in, while **authorship attribution** is the task of determining a text's author, relevant to both humanistic and forensic analysis.

But what makes classification so important is that **language modeling** can also be viewed as classification: each word can be thought of as a class, and so predicting the next word is classifying the context-so-far into a class for each next word. As we'll see, this intuition underlies large language models.

The algorithm for classification we introduce in this chapter, logistic regression, is equally important, in a number of ways. First, logistic regression has a close relationship with neural networks. As we will see in Chapter 6, a neural network can be viewed as a series of logistic regression classifiers stacked on top of each other. Second, logistic regression introduces ideas that are fundamental to neural networks and language models, like the **sigmoid** and **softmax** functions, the **logit**, and the key **gradient descent** algorithm for learning. Finally, logistic regression is also one of the most important analytic tools in the social and natural sciences.

sigmoid
softmax
logit

4.1 Machine learning and classification

The goal of classification is to take a single input (we call each input an **observation**), extract some useful **features** or properties of the input, and thereby **classify** the observation into one of a set of discrete classes. We'll call the input x, and say that the output comes from a fixed set of output classes $Y = \{y_1, y_2, ..., y_M\}$. Our goal is return a predicted class $\hat{y} \in Y$. The **hat** or **circumflex** notation $\hat{y}$ is used to refer to an estimated or predicted value. Sometimes you'll see the output classes referred to as the set C instead of Y.

observation
hat

For sentiment analysis, the input x might be a review, or some other text. And the output set Y might be the set:

$$\{\texttt{positive}, \texttt{negative}\}$$

or the set

$$\{0, 1\}$$

For language id, the input might be a text that we need to know what language it was written in, and the output set Y is the set of languages, i.e.,

$$Y = \{\texttt{Abkhaz}, \texttt{Ainu}, \texttt{Albanian}, \texttt{Amharic}, ...\texttt{Zulu}, \texttt{Zuñi}\}$$

There are many ways to do classification. One method is to use rules handwritten by humans. For example, we might have a rule like:

```
If the word ‘‘love’’ appears in x, and it’s not preceded by the
word ‘‘don’t", classify as positive
```

Handwritten rules can be components of modern NLP systems, such as the handwritten lists of positive and negative words that can be used in sentiment analysis, as we'll see below. But rules can be fragile, as situations or data change over time, and for many tasks there are complex interactions between different features (like the example of negation with "don't" in the rule above), so it can be quite hard for humans to come up with rules that are successful over many situations.

Another method that we will introduce later is to ask a large language model (of the type we will introduce in Chapter 7) by prompting the model to give a label to some text. Prompting can be powerful, but again has weaknesses: language models often hallucinate, and may not be able to explain why they chose the class they did.

For these reasons the most common way to do classification is to use **supervised machine learning**. Supervised machine learning is a paradigm in which, in

supervised machine learning

addition to the input and the set of output classes, we have a **labeled training set** and a **learning algorithm**. We talked about training sets in Chapter 3 as a locus for computing n-gram statistics. But in supervised machine learning the training set is **labeled**, meaning that it contains a set of input observations, each observation associated with the correct output (a 'supervision signal'). We can generally refer to a training set of m input/output pairs, where each input x is a text, in the case of text classification, and each is hand-labeled with an associated class (the correct label):.

$$\text{training set: } \{(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)}),\ldots,(x^{(m)},y^{(m)})\} \tag{4.1}$$

We'll use superscripts in parentheses to refer to individual observations or instances in the training set. So for sentiment classification, a training set might be a set of sentences or other texts, each with their correct sentiment label.

Our goal is to learn from this training set a classifier that is capable of mapping from a **new** input x to its correct class $y \in Y$. It does this by learn to find features in these training sentences (perhaps words like "awesome" or "awful"). **Probabilistic classifiers** are the subset of machine learning classifiers that in addition to giving an answer (which class is this observation in?), additionally will tell us the *probability* of the observation being in the class. This full distribution over the classes can be useful information for downstream decisions; avoiding making discrete decisions early on can be useful when combining systems.

There are many algorithms for achieving this supervised machine learning task, (naive Bayes, support vector machines, neural networks, fine-tuned language models), but logistic regression has the advantages we discussed above and so we'll introduce it! Any machine learning classifier thus 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 $P(y|x)$. 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.

At the highest level, logistic regression, and really any supervised machine learning classifier, has two phases

training: We train the system (in the case of logistic regression that means training the weights w and b, introduced below) using stochastic gradient descent and the cross-entropy loss.

test: Given a test example x we compute the probability $P(y_i|x)$ of each class y_i, and return the higher probability label $y = 1$ or $y = 0$.

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 first describe this special case of logistic regression in the next few sections, beginning with the **sigmoid** function, and then turn to **multinomial logistic regression** for more than two classes and the use of the **softmax** function in Section 4.4.

4.2 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.

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b \tag{4.2}$$

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 $\mathbf{a} \cdot \mathbf{b}$, is the sum of the products of the corresponding elements of each vector. (Notice that we represent vectors using the boldface notation **b**). Thus the following is an equivalent formation to Eq. 4.2:

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{4.3}$$

But note that nothing in Eq. 4.3 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 ∞.

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. 4.1:

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-z)} \tag{4.4}$$

(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

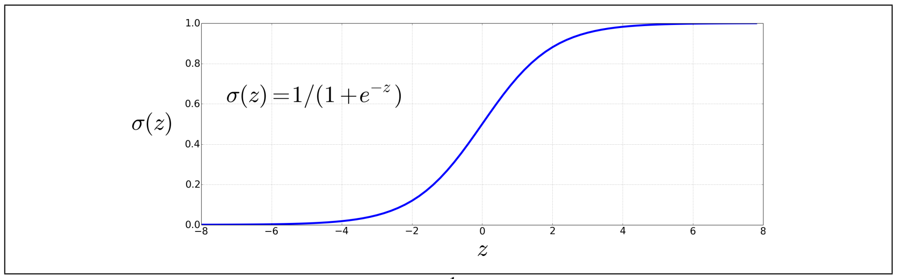


Figure 4.1 The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

$(0,1)$, 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 4.15 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:

$$\begin{aligned} P(y=1) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ P(y=0) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= 1-\frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ &= \frac{\exp(-(\mathbf{w}\cdot\mathbf{x}+b))}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \end{aligned} \tag{4.5}$$

The sigmoid function has the property

$$1-\sigma(x) = \sigma(-x) \tag{4.6}$$

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. 4.3, 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

$$\text{logit}(p) = \sigma^{-1}(p) = \ln\frac{p}{1-p} \tag{4.7}$$

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.

4.3 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.

4.3.1 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 $x_1 \dots x_6$ of the input shown in the following table; Fig. 4.2 shows features in a sample mini test document.

Var	Definition	Value in Fig. 4.2
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+punctuation count of doc)	$\ln(66) = 4.19$

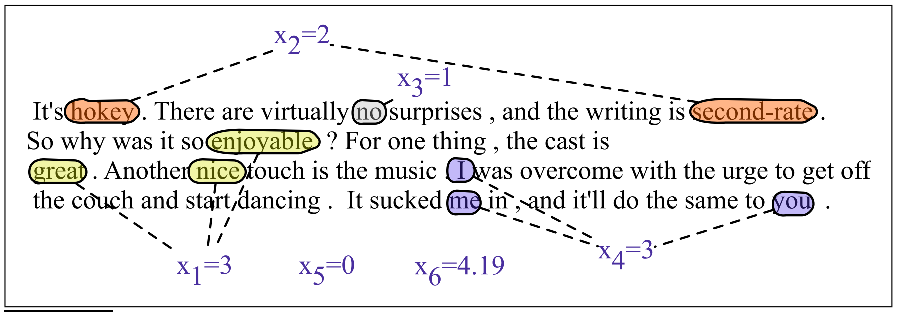

x2=2
x3=1
It's hokey. There are virtually no surprises , and the writing is second-rate.
So why was it so enjoyable ? For one thing , the cast is
great . Another nice touch is the music . I was overcome with the urge to get off
the couch and start dancing . It sucked me in , and it'll do the same to you .
x1=3
x5=0
x6=4.19
x4=3

Figure 4.2 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. 4.5:

$$\begin{aligned} P(+|x) = P(y=1|x) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \sigma([2.5,-5.0,-1.2,0.5,2.0,0.7]\cdot[3,2,1,3,0,4.19]+0.1) \\ &= \sigma(.833) \\ &= 0.70 \\ P(-|x) = P(y=0|x) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= 0.30 \end{aligned} \tag{4.8}$$

4.3.2 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.

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

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 5 and Chapter 6.

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
z-score

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:

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

$$x'_i = \frac{x_i-\mu_i}{\sigma_i} \tag{4.9}$$

normalize

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

$$x'_i = \frac{x_i - \min(x_i)}{\max(x_i)-\min(x_i)} \tag{4.10}$$

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.

4.3.3 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 3, 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 $\mathbf{x}^{(i)}$, $1 \le i \le m$. (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:

$$\begin{aligned} \textbf{foreach} \quad & x^{(i)} \ \text{ in input } [x^{(1)},x^{(2)},...,x^{(m)}] \\ & y^{(i)} = \sigma(\mathbf{w}\cdot\mathbf{x^{(i)}}+b) \end{aligned} \tag{4.11}$$

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

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

But it turns out that we can slightly modify our original equation Eq. 4.5 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:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_f^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_f^{(2)} \\ x_1^{(3)} & x_2^{(3)} & \dots & x_f^{(3)} \\ \dots \end{bmatrix} \tag{4.12}$$

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:

$$\hat{\mathbf{y}} = \sigma(\mathbf{X}\mathbf{w} + \mathbf{b}) \tag{4.13}$$

You should convince yourself that Eq. 4.13 computes the same thing as our for-loop in Eq. 4.11. For example $\hat{y}^{(1)}$, the first entry of the output vector $\mathbf{y}$, will correctly be:

$$\hat{y}^{(1)} = [x_1^{(1)}, x_2^{(1)}, ..., x_f^{(1)}] \cdot [w_1, w_2, ..., w_f] + b \tag{4.14}$$

Note that we had to reorder $\mathbf{X}$ and $\mathbf{w}$ from the order they appeared in in Eq. 4.5 to make the multiplications come out properly. Here is Eq. 4.13 again with the shapes shown:

$$\begin{array}{ccccccc} \hat{\mathbf{y}} & = & \sigma(\mathbf{X} & & \mathbf{w} & + & \mathbf{b}) \\ (m \times 1) & & (m \times f) & & (f \times 1) & & (m \times 1) \end{array} \tag{4.15}$$

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 (as $\hat{\mathbf{y}} = \sigma(\mathbf{w}\mathbf{X} + \mathbf{b})$) 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.

4.4 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). Or, for large language models, we'll be predicting the next word out of the $|V|$ possible words in the vocabulary, so it's $|V|$-way classification.

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 **y** for each input **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 **y** to be 0, i.e., $y_c = 1$ and $y_j = 0 \quad \forall j \neq c$. A vector like this **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 $P(y_k = 1|\mathbf{x})$.

4.4.1 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 **z** of dimensionality K, the softmax is defined as:

$$\text{softmax}(z_i) \;=\; \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \quad 1 \leq i \leq K \tag{4.16}$$

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

$$\text{softmax}(\mathbf{z}) \;=\; \left[\frac{\exp(z_1)}{\sum_{i=1}^{K} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K} \exp(z_i)}, ..., \frac{\exp(z_K)}{\sum_{i=1}^{K} \exp(z_i)} \right] \tag{4.17}$$

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(**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 **z**, the vector of scores that is the input to the softmax, as **logits** (see Eq. 4.7).

4.4.2 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 **w** and an input vector **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:

$$P(y_k = 1|\mathbf{x}) \;=\; \frac{\exp(\mathbf{w}_k \cdot \mathbf{x} + b_k)}{\displaystyle\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + b_j)} \tag{4.18}$$

The form of Eq. 4.18 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:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{4.19}$$

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 (Doumbouya et al., 2025).

Fig. 4.3 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.

4.4.3 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 6:

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

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 $f_5(x)$ 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.

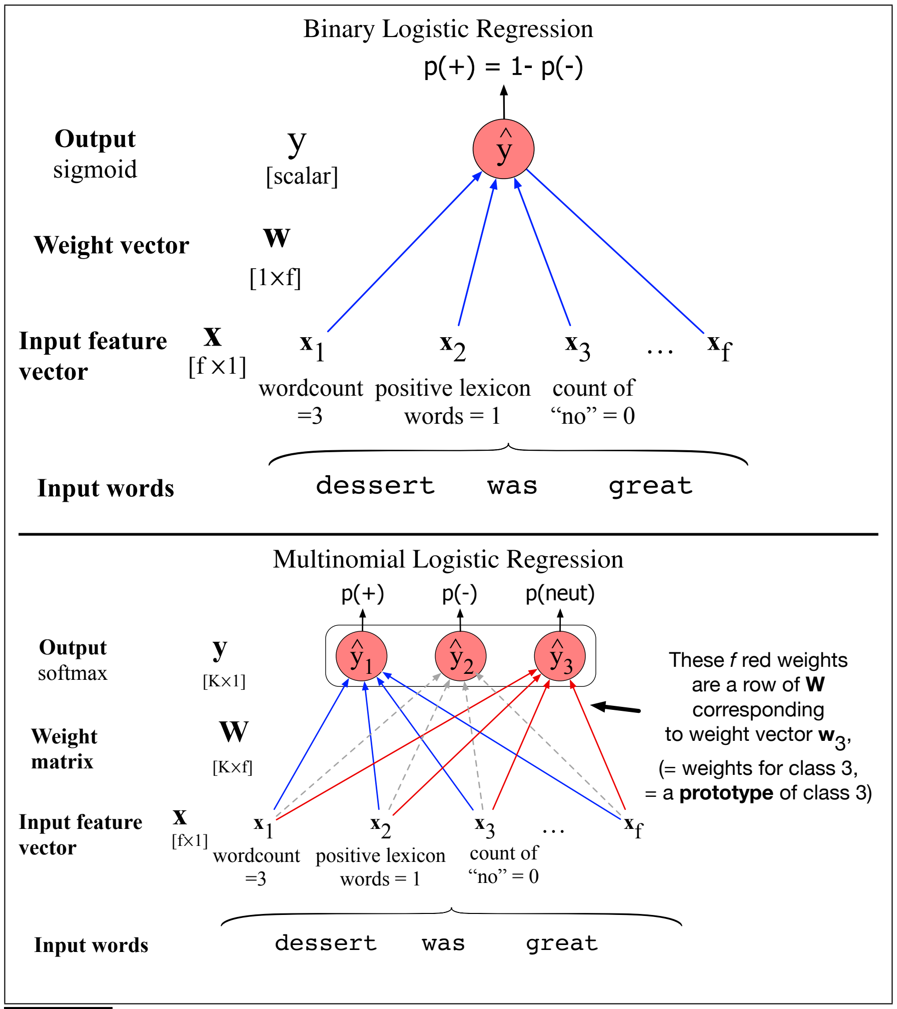


Figure 4.3 Binary versus multinomial logistic regression. Binary logistic regression uses a single weight vector **w**, and has a scalar output $\hat{y}$. In multinomial logistic regression we have K separate weight vectors corresponding to the K classes, all packed into a single weight matrix **W**, and a vector output $\hat{\mathbf{y}}$. We omit the biases from both figures for clarity.

4.5 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. 4.5 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 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,

loss

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.

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 4.8.

4.6 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:

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y \tag{4.20}$$

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. 4.21 simplifies to $\hat{y}$; if $y = 0$, Eq. 4.21 simplifies to $1 - \hat{y}$):

$$p(y|x) = \hat{y}^{y} (1 - \hat{y})^{1-y} \tag{4.21}$$

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:

$$\begin{aligned} \log p(y|x) &= \log \left[\hat{y}^{y} (1 - \hat{y})^{1-y}\right] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \end{aligned} \tag{4.22}$$

Eq. 4.22 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. 4.22. The result is the cross-entropy loss L_{CE}:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \tag{4.23}$$

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

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1 - y) \log(1 - \sigma(\mathbf{w} \cdot \mathbf{x} + b))] \tag{4.24}$$

Let's see if this loss function does the right thing for our example from Fig. 4.2. 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. 4.2 is positive, i.e., $y = 1$. In this case our model is doing well, since from Eq. 4.8 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. 4.24, 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. 4.2 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. 4.8 into Eq. 4.24, 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. 4.22 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.

4.7 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. 4.25 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:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)};\theta),y^{(i)}) \tag{4.25}$$

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. 4.4.

Given a random initialization of w at some value w^1, and assuming the loss function L happened to have the shape in Fig. 4.4, 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.

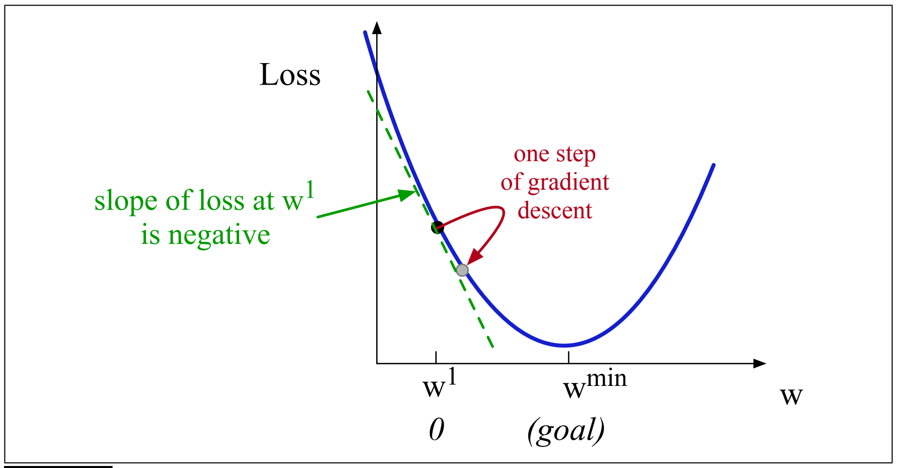

Loss
slope of loss at w^1 is negative
one step of gradient descent
w^1
0
w^{min}
(goal)
w

Figure 4.4 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. 4.4, we can informally think of the gradient as the slope. The dotted line in Fig. 4.4 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):

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x;w), y) \tag{4.26}$$

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

$$\nabla L(f(x;\theta), y) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta), y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta), y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta), y) \\ \frac{\partial}{\partial b} L(f(x;\theta), y) \end{bmatrix} \tag{4.27}$$

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

$$\theta^{t+1} = \theta^t - \eta \nabla L(f(x;\theta), y) \tag{4.28}$$

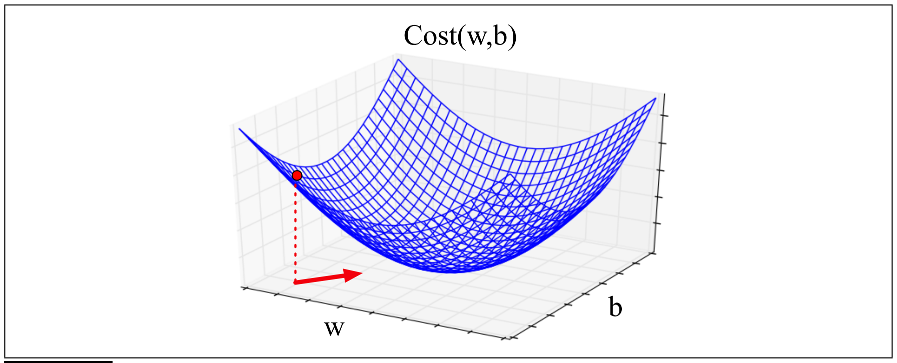


Figure 4.5 Visualization of the gradient vector at the red point in two dimensions w and b, showing a red arrow in the x-y plane pointing in the direction we will go to look for the minimum: the opposite direction of the gradient (recall that the gradient points in the direction of increase not decrease).

4.7.1 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:

$$L_{\text{CE}}(\hat{y},y) = -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \quad (4.29)$$

It turns out that the derivative of this function for one observation vector x is Eq. 4.30 (the interested reader can see Section 4.15 for the derivation of this equation):

$$\begin{aligned}\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j \\ &= (\hat{y}-y)x_j \end{aligned} \quad (4.30)$$

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

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = -(y-\hat{y})x_j \quad (4.31)$$

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.

4.7.2 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 4.7.4 we'll discuss other versions of gradient descent that batch many examples at once. Fig. 4.6 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.

We'll discuss hyperparameters in more detail in Chapter 6, 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.

4.7.3 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. 4.2 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:

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

```
function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, $x$, $y$) returns $\theta$
     # where: L is the loss function
     #       f is a function parameterized by $\theta$
     #       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 $\theta$ to maximize loss?
      3. $\theta \leftarrow \theta - \eta\, g$                 # Go the other way instead
return $\theta$
```

Figure 4.6 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 6.

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:

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

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.

4.7.4 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 4.6 and the gradient in Section 4.7.1. Let's extend the cross-entropy loss for one example from Eq. 4.23 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:

$$\begin{aligned}\log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) \\ &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \\ &= -\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \end{aligned} \tag{4.32}$$

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

$$\begin{aligned} Cost(\hat{y}, y) &= \frac{1}{m}\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b) + (1-y^{(i)})\log\left(1-\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b)\right) \end{aligned} \tag{4.33}$$

The mini-batch gradient is the average of the individual gradients from Eq. 4.30:

$$\frac{\partial Cost(\hat{y}, y)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left[\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b) - y^{(i)}\right] x_j^{(i)} \tag{4.34}$$

Instead of using the sum notation, we can more efficiently compute the gradient in its matrix form, following the vectorization we saw on page 9, 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:

$$\begin{aligned}\frac{\partial Cost(\hat{y},y)}{\partial \mathbf{w}} &= \frac{1}{m}(\hat{\mathbf{y}}-\mathbf{y})^\intercal \mathbf{X} \\ &= \frac{1}{m}(\sigma(\mathbf{X}\mathbf{w}+\mathbf{b})-\mathbf{y})^\intercal \mathbf{X}\end{aligned} \quad (4.35)$$

4.8 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. 4.23) is:

$$L_{\text{CE}}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y}+(1-y)\log(1-\hat{y})] \quad (4.36)$$

The loss function for multinomial logistic regression generalizes the two terms in Eq. 4.36 (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. 4.37). This turns out to be just the negative log probability of the correct class c (Eq. 4.38):

$$\begin{aligned}L_{\text{CE}}(\hat{\mathbf{y}},\mathbf{y}) &= -\sum_{k=1}^{K} y_k\log\hat{y}_k && (4.37)\\ &= -\log\hat{y}_c, \quad \text{(where } c \text{ is the correct class)} && (4.38)\\ &= -\log\hat{p}(y_c=1|\mathbf{x}) \quad \text{(where } c \text{ is the correct class)} \\ &= -\log\frac{\exp(\mathbf{w_c}\cdot\mathbf{x}+b_c)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)} \quad (c \text{ is the correct class}) && (4.39)\end{aligned}$$

How did we get from Eq. 4.37 to Eq. 4.38? 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. 4.37 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. 4.38 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. 4.30. 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:

$$\begin{aligned}
\frac{\partial L_{\text{CE}}}{\partial w_{k,i}} &= -(y_k - \hat{y}_k)x_i \\
&= -(y_k - p(y_k = 1|x))x_i \\
&= -\left(y_k - \frac{\exp(\mathbf{w_k} \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w_j} \cdot \mathbf{x} + b_j)}\right) x_i
\end{aligned} \tag{4.40}$$

We'll return to this case of the gradient for softmax regression when we introduce neural networks in Chapter 6, and at that time we'll also discuss the derivation of this gradient in equations Eq. **??**–Eq. **??**.

4.9 Evaluation: Precision, Recall, F-measure

To introduce the methods for evaluating text classification, let's first consider some simple binary *detection* tasks. For example, in spam detection, our goal is to label every text as being in the spam category ("positive") or not in the spam category ("negative"). For each item (email document) we therefore need to know whether our system called it spam or not. We also need to know whether the email is actually spam or not, i.e. the human-defined labels for each document that we are trying to match. We will refer to these human labels as the **gold labels**.

gold labels

Or imagine you're the CEO of the *Delicious Pie Company* and you need to know what people are saying about your pies on social media, so you build a system that detects tweets concerning Delicious Pie. Here the positive class is tweets about Delicious Pie and the negative class is all other tweets.

In both cases, we need a metric for knowing how well our spam detector (or pie-tweet-detector) is doing. To evaluate any system for detecting things, we start by building a **confusion matrix** like the one shown in Fig. 4.7. A confusion matrix is a table for visualizing how an algorithm performs with respect to the human gold labels, using two dimensions (system output and gold labels), and each cell labeling a set of possible outcomes. In the spam detection case, for example, true positives are documents that are indeed spam (indicated by human-created gold labels) that our system correctly said were spam. False negatives are documents that are indeed spam but our system incorrectly labeled as non-spam.

confusion matrix

To the bottom right of the table is the equation for *accuracy*, which asks what percentage of all the observations (for the spam or pie examples that means all emails or tweets) our system labeled correctly. Although accuracy might seem a natural metric, we generally don't use it for text classification tasks. That's because accuracy doesn't work well when the classes are unbalanced (as indeed they are with spam, which is a large majority of email, or with tweets, which are mainly not about pie).

To make this more explicit, imagine that we looked at a million tweets, and let's say that only 100 of them are discussing their love (or hatred) for our pie, while the other 999,900 are tweets about something completely unrelated. Imagine a simple classifier that stupidly classified every tweet as "not about pie". This classifier would have 999,900 true negatives and only 100 false negatives for an accuracy of 999,900/1,000,000 or 99.99%! What an amazing accuracy level! Surely we should be happy with this classifier? But of course this fabulous 'no pie' classifier would be completely useless, since it wouldn't find a single one of the customer comments we are looking for. In other words, accuracy is not a good metric when the goal is

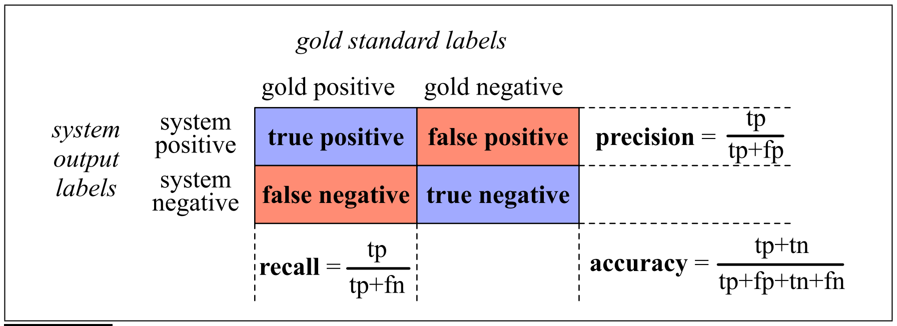


Figure 4.7 A confusion matrix for visualizing how well a binary classification system performs against gold standard labels.

to discover something that is rare, or at least not completely balanced in frequency, which is a very common situation in the world.

precision That's why instead of accuracy we generally turn to two other metrics shown in Fig. 4.7: **precision** and **recall**. **Precision** measures the percentage of the items that the system detected (i.e., the system labeled as positive) that are in fact positive (i.e., are positive according to the human gold labels). Precision is defined as

$$\textbf{Precision} = \frac{\text{true positives}}{\text{true positives + false positives}}$$

recall **Recall** measures the percentage of items actually present in the input that were correctly identified by the system. Recall is defined as

$$\textbf{Recall} = \frac{\text{true positives}}{\text{true positives + false negatives}}$$

Precision and recall will help solve the problem with the useless "nothing is pie" classifier. This classifier, despite having a fabulous accuracy of 99.99%, has a terrible recall of 0 (since there are no true positives, and 100 false negatives, the recall is 0/100). You should convince yourself that the precision at finding relevant tweets is equally problematic. Thus precision and recall, unlike accuracy, emphasize true positives: finding the things that we are supposed to be looking for.

F-measure There are many ways to define a single metric that incorporates aspects of both precision and recall. The simplest of these combinations is the **F-measure** (van Rijsbergen, 1975) , defined as:

$$F_\beta = \frac{(\beta^2+1)PR}{\beta^2 P + R}$$

The β parameter differentially weights the importance of recall and precision, based perhaps on the needs of an application. Values of $\beta > 1$ favor recall, while values of $\beta < 1$ favor precision. When $\beta = 1$, precision and recall are equally balanced; this is the most frequently used metric, and is called $\mathrm{F}_{\beta=1}$ or just F_1:

F1

$$\mathrm{F}_1 = \frac{2PR}{P+R} \tag{4.41}$$

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of reciprocals:

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, ..., a_n) = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \frac{1}{a_3} + ... + \frac{1}{a_n}} \tag{4.42}$$

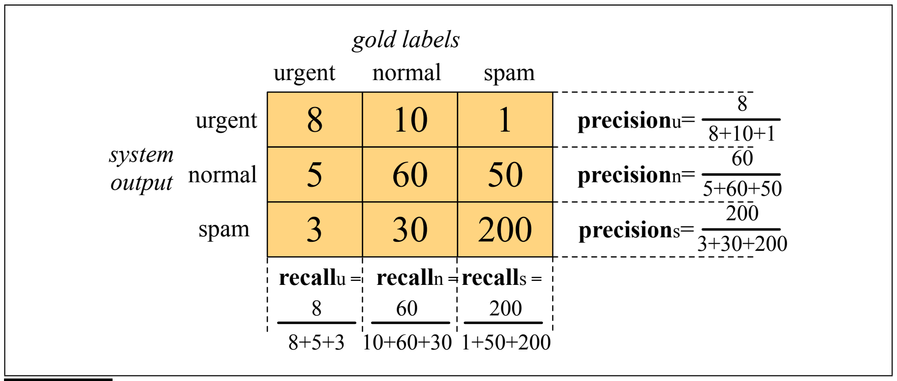
and hence F-measure is

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{4.43}$$

Harmonic mean is used because the harmonic mean of two values is closer to the minimum of the two values than the arithmetic mean is. Thus it weighs the lower of the two numbers more heavily, which is more conservative in this situation.

4.9.1 Evaluating with more than two classes

Up to now we have been describing text classification tasks with only two classes. But lots of classification tasks in language processing have more than two classes. For sentiment analysis we generally have 3 classes (positive, negative, neutral) and even more classes are common for tasks like part-of-speech tagging, word sense disambiguation, semantic role labeling, emotion detection, and so on. Luckily the naive Bayes algorithm is already a multi-class classification algorithm.



		gold labels urgent	normal	spam	
system output	urgent	8	10	1	**precision**u= $\frac{8}{8+10+1}$
	normal	5	60	50	**precision**n= $\frac{60}{5+60+50}$
	spam	3	30	200	**precision**s= $\frac{200}{3+30+200}$
		recallu = $\frac{8}{8+5+3}$	**recall**n = $\frac{60}{10+60+30}$	**recall**s = $\frac{200}{1+50+200}$	

Figure 4.8 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2.

But we'll need to slightly modify our definitions of precision and recall. Consider the sample confusion matrix for a hypothetical 3-way *one-of* email categorization decision (urgent, normal, spam) shown in Fig. 4.8. The matrix shows, for example, that the system mistakenly labeled one spam document as urgent, and we have shown how to compute a distinct precision and recall value for each class. In order to derive a single metric that tells us how well the system is doing, we can combine these values in two ways. In **macroaveraging**, we compute the performance for each class, and then average over classes. In **microaveraging**, we collect the decisions for all classes into a single confusion matrix, and then compute precision and recall from that table. Fig. 4.9 shows the confusion matrix for each class separately, and shows the computation of microaveraged and macroaveraged precision.

macroaveraging
microaveraging

As the figure shows, a microaverage is dominated by the more frequent class (in this case spam), since the counts are pooled. The macroaverage better reflects the statistics of the smaller classes, and so is more appropriate when performance on all the classes is equally important.

Class 1: Urgent

	true urgent	true not
system urgent	8	11
system not	8	340

$$\text{precision} = \frac{8}{8+11} = .42$$

Class 2: Normal

	true normal	true not
system normal	60	55
system not	40	212

$$\text{precision} = \frac{60}{60+55} = .52$$

Class 3: Spam

	true spam	true not
system spam	200	33
system not	51	83

$$\text{precision} = \frac{200}{200+33} = .86$$

Pooled

	true yes	true no
system yes	268	99
system no	99	635

$$\begin{array}{c}\text{microaverage}\\\text{precision}\end{array} = \frac{268}{268+99} = \mathbf{.73}$$

$$\begin{array}{c}\text{macroaverage}\\\text{precision}\end{array} = \frac{.42+.52+.86}{3} = \mathbf{.60}$$

Figure 4.9 Separate confusion matrices for the 3 classes from the previous figure, showing the pooled confusion matrix and the microaveraged and macroaveraged precision.

4.10 Test sets and Cross-validation

The training and testing procedure for text classification follows what we saw with language modeling (Section **??**): we use the training set to train the model, then use the **development test set** (also called a **devset**) to perhaps tune some parameters, and in general decide what the best model is. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

development test set
devset

While the use of a devset avoids overfitting the test set, having a fixed training set, devset, and test set creates another problem: in order to save lots of data for training, the test set (or devset) might not be large enough to be representative. Wouldn't it be better if we could somehow use all our data for training and still use all our data for test? We can do this by **cross-validation**.

cross-validation

In cross-validation, we choose a number k, and partition our data into k disjoint subsets called **folds**. Now we choose one of those k folds as a test set, train our classifier on the remaining $k-1$ folds, and then compute the error rate on the test set. Then we repeat with another fold as the test set, again training on the other $k-1$ folds. We do this sampling process k times and average the test set error rate from these k runs to get an average error rate. If we choose $k = 10$, we would train 10 different models (each on 90% of our data), test the model 10 times, and average these 10 values. This is called **10-fold cross-validation**.

folds
10-fold cross-validation

The only problem with cross-validation is that because all the data is used for testing, we need the whole corpus to be blind; we can't examine any of the data to suggest possible features and in general see what's going on, because we'd be peeking at the test set, and such cheating would cause us to overestimate the performance of our system. However, looking at the corpus to understand what's going on is important in designing NLP systems! What to do? For this reason, it is common to create a fixed training set and test set, then do 10-fold cross-validation inside the training set, but compute error rate the normal way in the test set, as shown in Fig. 4.10.

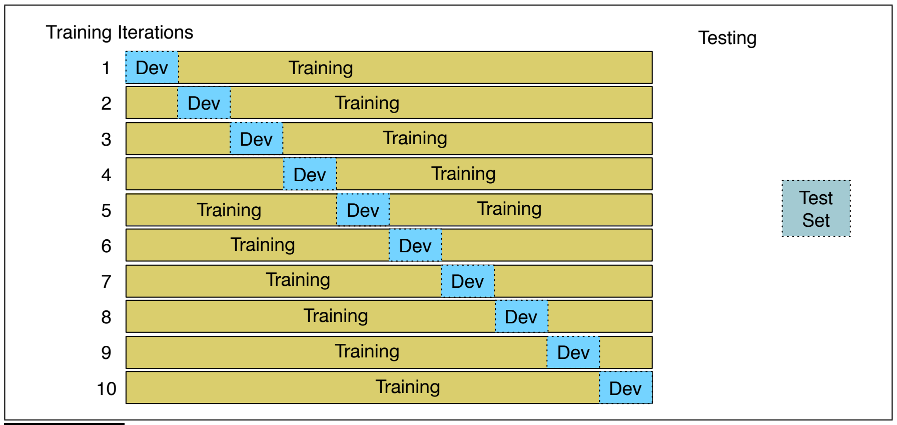


Figure 4.10 10-fold cross-validation

4.11 Statistical Significance Testing

In building systems we often need to compare the performance of two systems. How can we know if the new system we just built is better than our old one? Or better than some other system described in the literature? This is the domain of statistical hypothesis testing, and in this section we introduce tests for statistical significance for NLP classifiers, drawing especially on the work of Dror et al. (2020) and Berg-Kirkpatrick et al. (2012).

Suppose we're comparing the performance of classifiers A and B on a metric M such as F_1, or accuracy. Perhaps we want to know if our new sentiment classifier A gets a higher F_1 score than our previous sentiment classifier B on a particular test set x. Let's call $M(A,x)$ the score that system A gets on test set x, and $\delta(x)$ the performance difference between A and B on x:

$$\delta(x) = M(A,x) - M(B,x) \tag{4.44}$$

effect size

We would like to know if $\delta(x) > 0$, meaning that our logistic regression classifier has a higher F_1 than our naive Bayes classifier on x. $\delta(x)$ is called the **effect size**; a bigger δ means that A seems to be way better than B; a small δ means A seems to be only a little better.

Why don't we just check if $\delta(x)$ is positive? Suppose we do, and we find that the F_1 score of A is higher than B's by .04. Can we be certain that A is better? We cannot! That's because A might just be accidentally better than B on this particular x. We need something more: we want to know if A's superiority over B is likely to hold again if we checked another test set x', or under some other set of circumstances.

In the paradigm of statistical hypothesis testing, we test this by formalizing two hypotheses.

$$\begin{aligned} H_0 &: \delta(x) \leq 0 \\ H_1 &: \delta(x) > 0 \end{aligned} \tag{4.45}$$

null hypothesis

The hypothesis H_0, called the **null hypothesis**, supposes that $\delta(x)$ is actually negative or zero, meaning that A is not better than B. We would like to know if we can confidently rule out this hypothesis, and instead support H_1, that A is better.

We do this by creating a random variable X ranging over all test sets. Now we ask how likely is it, if the null hypothesis H_0 was correct, that among these test sets we would encounter the value of $\delta(x)$ that we found, if we repeated the experiment a great many times. We formalize this likelihood as the **p-value**: the probability, assuming the null hypothesis H_0 is true, of seeing the $\delta(x)$ that we saw or one even greater

p-value

$$P(\delta(X) \geq \delta(x)|H_0 \text{ is true}) \tag{4.46}$$

So in our example, this p-value is the probability that we would see $\delta(x)$ assuming A is **not** better than B. If $\delta(x)$ is huge (let's say A has a very respectable F_1 of .9 and B has a terrible F_1 of only .2 on x), we might be surprised, since that would be extremely unlikely to occur if H_0 were in fact true, and so the p-value would be low (unlikely to have such a large δ if A is in fact not better than B). But if $\delta(x)$ is very small, it might be less surprising to us even if H_0 were true and A is not really better than B, and so the p-value would be higher.

A very small p-value means that the difference we observed is very unlikely under the null hypothesis, and we can reject the null hypothesis. What counts as very small? It is common to use values like .05 or .01 as the thresholds. A value of .01 means that if the p-value (the probability of observing the δ we saw assuming H_0 is true) is less than .01, we reject the null hypothesis and assume that A is indeed better than B. We say that a result (e.g., "A is better than B") is **statistically significant** if the δ we saw has a probability that is below the threshold and we therefore reject this null hypothesis.

statistically significant

How do we compute this probability we need for the p-value? In NLP we generally don't use simple parametric tests like t-tests or ANOVAs that you might be familiar with. Parametric tests make assumptions about the distributions of the test statistic (such as normality) that don't generally hold in our cases. So in NLP we usually use non-parametric tests based on sampling: we artificially create many versions of the experimental setup. For example, if we had lots of different test sets x' we could just measure all the $\delta(x')$ for all the x'. That gives us a distribution. Now we set a threshold (like .01) and if we see in this distribution that 99% or more of those deltas are smaller than the delta we observed, i.e., that p-value(x)—the probability of seeing a $\delta(x)$ as big as the one we saw—is less than .01, then we can reject the null hypothesis and agree that $\delta(x)$ was a sufficiently surprising difference and A is really a better algorithm than B.

There are two common non-parametric tests used in NLP: **approximate randomization** (Noreen, 1989) and the **bootstrap test**. We will describe bootstrap below, showing the paired version of the test, which again is most common in NLP. **Paired** tests are those in which we compare two sets of observations that are aligned: each observation in one set can be paired with an observation in another. This happens naturally when we are comparing the performance of two systems on the same test set; we can pair the performance of system A on an individual observation x_i with the performance of system B on the same x_i.

approximate randomization

paired

4.11.1 The Paired Bootstrap Test

bootstrap test

The **bootstrap test** (Efron and Tibshirani, 1993) can apply to any metric; from precision, recall, or F1 to the BLEU metric used in machine translation. The word **bootstrapping** refers to repeatedly drawing large numbers of samples with replacement (called **bootstrap samples**) from an original set. The intuition of the bootstrap test is that we can create many virtual test sets from an observed test set by repeatedly sampling from it. The method only makes the assumption that the sample is representative of the population.

bootstrapping

Consider a tiny text classification example with a test set x of 10 documents. The first row of Fig. 4.11 shows the results of two classifiers (A and B) on this test set. Each document is labeled by one of the four possibilities (A and B both right, both wrong, A right and B wrong, A wrong and B right). A slash through a letter ($\cancel{\text{B}}$) means that that classifier got the answer wrong. On the first document both A and B get the correct class (AB), while on the second document A got it right but B got it wrong (A$\cancel{\text{B}}$). If we assume for simplicity that our metric is accuracy, A has an accuracy of .70 and B of .50, so $\delta(x)$ is .20.

Now we create a large number b (perhaps 10^5) of virtual test sets $x^{(i)}$, each of size $n = 10$. Fig. 4.11 shows a couple of examples. To create each virtual test set $x^{(i)}$, we repeatedly ($n = 10$ times) select a cell from row x with replacement. For example, to create the first cell of the first virtual test set $x^{(1)}$, if we happened to randomly select the second cell of the x row, we would copy the value A$\cancel{\text{B}}$ into our new cell, and move on to create the second cell of $x^{(1)}$, each time sampling (randomly choosing) from the original x with replacement.

	1	2	3	4	5	6	7	8	9	10	A%	B%	$\delta()$
x	AB	A$\cancel{\text{B}}$	AB	$\cancel{\text{A}}$B	A$\cancel{\text{B}}$	$\cancel{\text{A}}$B	A$\cancel{\text{B}}$	AB	$\cancel{\text{A}}\cancel{\text{B}}$	A$\cancel{\text{B}}$	.70	.50	.20
$x^{(1)}$	A$\cancel{\text{B}}$	AB	A$\cancel{\text{B}}$	$\cancel{\text{A}}$B	$\cancel{\text{A}}$B	A$\cancel{\text{B}}$	$\cancel{\text{A}}$B	AB	$\cancel{\text{A}}\cancel{\text{B}}$	AB	.60	.60	.00
$x^{(2)}$	A$\cancel{\text{B}}$	AB	$\cancel{\text{A}}\cancel{\text{B}}$	$\cancel{\text{A}}$B	$\cancel{\text{A}}$B	AB	$\cancel{\text{A}}$B	A$\cancel{\text{B}}$	AB	AB	.60	.70	-.10
...													
$x^{(b)}$													

Figure 4.11 The paired bootstrap test: Examples of b pseudo test sets $x^{(i)}$ being created from an initial true test set x. Each pseudo test set is created by sampling $n = 10$ times with replacement; thus an individual sample is a single cell, a document with its gold label and the correct or incorrect performance of classifiers A and B. Of course real test sets don't have only 10 examples, and b needs to be large as well.

Now that we have the b test sets, providing a sampling distribution, we can do statistics on how often A has an accidental advantage. There are various ways to compute this advantage; here we follow the version laid out in Berg-Kirkpatrick et al. (2012). Assuming H_0 (A isn't better than B), we would expect that $\delta(X)$, estimated over many test sets, would be zero or negative; a much higher value would be surprising, since H_0 specifically assumes A isn't better than B. To measure exactly how surprising our observed $\delta(x)$ is, we would in other circumstances compute the p-value by counting over many test sets how often $\delta(x^{(i)})$ exceeds the expected zero value by $\delta(x)$ or more:

$$\text{p-value}(x) = \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) - \delta(x) \geq 0\right)$$

(We use the notation $\mathbb{1}(x)$ to mean "1 if x is true, and 0 otherwise".) However, although it's generally true that the expected value of $\delta(X)$ over many test sets, (again assuming A isn't better than B) is 0, this **isn't** true for the bootstrapped test sets we created. That's because we didn't draw these samples from a distribution with 0 mean; we happened to create them from the original test set x, which happens to be biased (by .20) in favor of A. So to measure how surprising is our observed $\delta(x)$, we actually compute the p-value by counting over many test sets how often

$\delta(x^{(i)})$ exceeds the expected value of $\delta(x)$ by $\delta(x)$ or more:

$$\begin{aligned}\text{p-value}(x) &= \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) - \delta(x) \geq \delta(x)\right) \\ &= \frac{1}{b}\sum_{i=1}^{b} \mathbb{1}\left(\delta(x^{(i)}) \geq 2\delta(x)\right) \qquad (4.47)\end{aligned}$$

So if for example we have 10,000 test sets $x^{(i)}$ and a threshold of .01, and in only 47 of the test sets do we find that A is accidentally better $\delta(x^{(i)}) \geq 2\delta(x)$, the resulting p-value of .0047 is smaller than .01, indicating that the delta we found, $\delta(x)$ is indeed sufficiently surprising and unlikely to have happened by accident, and we can reject the null hypothesis and conclude A is better than B.

```
function BOOTSTRAP(test set $x$, num of samples $b$) returns p-value($x$)

 Calculate $\delta(x)$   # how much better does algorithm A do than B on $x$
 $s$ = 0
 for $i$ = 1 to $b$ do
     for $j$ = 1 to $n$ do    # Draw a bootstrap sample $x^{(i)}$ of size n
         Select a member of $x$ at random and add it to $x^{(i)}$
     Calculate $\delta(x^{(i)})$   # how much better does algorithm A do than B on $x^{(i)}$
     $s \leftarrow s + 1$ if $\delta(x^{(i)}) \geq 2\delta(x)$
 p-value($x$) $\approx \frac{s}{b}$   # on what % of the b samples did algorithm A beat expectations?
 return p-value($x$)    # if very few did, our observed $\delta$ is probably not accidental
```

Figure 4.12 A version of the paired bootstrap algorithm after Berg-Kirkpatrick et al. (2012).

The full algorithm for the bootstrap is shown in Fig. 4.12. It is given a test set x, a number of samples b, and counts the percentage of the b bootstrap test sets in which $\delta(x^{(i)}) > 2\delta(x)$. This percentage then acts as a one-sided empirical p-value.

4.12 Avoiding Harms in Classification

It is important to avoid harms that may result from classifiers, harms that exist both for naive Bayes classifiers and for the other classification algorithms we introduce in later chapters.

representational harms

One class of harms is **representational harms** (Crawford 2017, Blodgett et al. 2020), harms caused by a system that demeans a social group, for example by perpetuating negative stereotypes about them. For example Kiritchenko and Mohammad (2018) examined the performance of 200 sentiment analysis systems on pairs of sentences that were identical except for containing either a common African American first name (like *Shaniqua*) or a common European American first name (like *Stephanie*), chosen from the Caliskan et al. (2017) study discussed in Chapter 5. They found that most systems assigned lower sentiment and more negative emotion to sentences with African American names, reflecting and perpetuating stereotypes that associate African Americans with negative emotions (Popp et al., 2003).

In other tasks classifiers may lead to both representational harms and other harms, such as silencing. For example the important text classification task of **toxicity detection** is the task of detecting hate speech, abuse, harassment, or other kinds of toxic language. While the goal of such classifiers is to help reduce societal harm, toxicity classifiers can themselves cause harms. For example, researchers have shown that some widely used toxicity classifiers incorrectly flag as being toxic sentences that are non-toxic but simply mention identities like women (Park et al., 2018), blind people (Hutchinson et al., 2020) or gay people (Dixon et al., 2018; Dias Oliva et al., 2021), or simply use linguistic features characteristic of varieties like African-American Vernacular English (Sap et al. 2019, Davidson et al. 2019). Such false positive errors could lead to the silencing of discourse by or about these groups.

toxicity detection

These model problems can be caused by biases or other problems in the training data; in general, machine learning systems replicate and even amplify the biases in their training data. But these problems can also be caused by the labels (for example due to biases in the human labelers), by the resources used (like lexicons, or model components like pretrained embeddings), or even by model architecture (like what the model is trained to optimize). While the mitigation of these biases (for example by carefully considering the training data sources) is an important area of research, we currently don't have general solutions. For this reason it's important, when introducing any NLP model, to study these kinds of factors and make them clear. One way to do this is by releasing a **model card** (Mitchell et al., 2019) for each version of a model. A model card documents a machine learning model with information like:

model card

- training algorithms and parameters
- training data sources, motivation, and preprocessing
- evaluation data sources, motivation, and preprocessing
- intended use and users
- model performance across different demographic or other groups and environmental situations

4.13 Interpreting models

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.

interpretable

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.

4.14 Advanced: 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. 4.25, resulting in the following loss for a batch of m examples (slightly rewritten from Eq. 4.25 to be maximizing log probability rather than minimizing loss, and removing the $\frac{1}{m}$ term which doesn't affect the argmax):

$$\hat{\theta} = \operatorname*{argmax}_{\theta} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta) \quad (4.48)$$

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. The higher the regularization strength parameter α, the lower the model's weights will be, reducing its reliance on the training data.

L2 regularization

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:

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2 \quad (4.49)$$

The L2 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)};\theta) \right] - \alpha \sum_{j=1}^{n} \theta_j^2 \tag{4.50}$$

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):

$$R(\theta) = ||\theta||_1 = \sum_{i=1}^{n} |\theta_i| \tag{4.51}$$

The L1 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)};\theta) \right] - \alpha \sum_{j=1}^{n} |\theta_j| \tag{4.52}$$

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

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2} \right) \tag{4.53}$$

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) \times \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2} \right) \tag{4.54}$$

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha \sum_{j=1}^{n} \theta_j^2 \tag{4.55}$$

which is in the same form as Eq. 4.50.

4.15 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 $\ln(x)$:

$$\frac{d}{dx}\ln(x) = \frac{1}{x} \tag{4.56}$$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z)) \tag{4.57}$$

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:

$$\frac{df}{dx} = \frac{du}{dv}\cdot\frac{dv}{dx} \tag{4.58}$$

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):

$$\begin{aligned}\frac{\partial L_{\text{CE}}}{\partial w_j} &= \frac{\partial}{\partial w_j} - [y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\ &= -\left[\frac{\partial}{\partial w_j} y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b) + \frac{\partial}{\partial w_j}(1-y)\log[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\right]\end{aligned} \tag{4.59}$$

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

$$\frac{\partial L_{\text{CE}}}{\partial w_j} = -\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b) - \frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)] \tag{4.60}$$

Rearranging terms:

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

And now plugging in the derivative of the sigmoid, and using the chain rule one more time, we end up with Eq. 4.61:

$$\begin{aligned}\frac{\partial L_{\text{CE}}}{\partial w_j} &= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\frac{\partial(\mathbf{w}\cdot\mathbf{x}+b)}{\partial w_j} \\ &= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j \\ &= -[y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j \\ &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j\end{aligned} \tag{4.61}$$

4.16 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.

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).

There are a variety of sources covering the many kinds of text classification tasks. For sentiment analysis see Pang and Lee (2008), and Liu and Zhang (2012). Stamatatos (2009) surveys authorship attribute algorithms. On language identification see Jauhiainen et al. (2019); Jaech et al. (2016) is an important early neural system. The task of newswire indexing was often used as a test case for text classification algorithms, based on the Reuters-21578 collection of newswire articles.

See Manning et al. (2008) and Aggarwal and Zhai (2012) on text classification; classification in general is covered in machine learning textbooks (Hastie et al. 2001, Witten and Frank 2005, Bishop 2006, Murphy 2012).

Non-parametric methods for computing statistical significance were used first in NLP in the MUC competition (Chinchor et al., 1993), and even earlier in speech recognition (Gillick and Cox 1989, Bisani and Ney 2004). Our description of the bootstrap draws on the description in Berg-Kirkpatrick et al. (2012). Recent work has focused on issues including multiple test sets and multiple metrics (Søgaard et al. 2014, Dror et al. 2017).

information gain

Feature selection is a method of removing features that are unlikely to generalize well. Features are generally ranked by how informative they are about the classification decision. A very common metric, **information gain**, tells us how many bits of information the presence of the word gives us for guessing the class. Other feature selection metrics include χ^2, pointwise mutual information, and GINI index; see Yang and Pedersen (1997) for a comparison and Guyon and Elisseeff (2003) for an introduction to feature selection.

Exercises

Aggarwal, C. C. and C. Zhai. 2012. A survey of text classification algorithms. In C. C. Aggarwal and C. Zhai, eds, *Mining text data*, 163–222. Springer.

Berg-Kirkpatrick, T., D. Burkett, and D. Klein. 2012. An empirical investigation of statistical significance in NLP. *EMNLP*.

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.

Bisani, M. and H. Ney. 2004. Bootstrap estimates for confidence intervals in ASR performance evaluation. *ICASSP*.

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

Blodgett, S. L., S. Barocas, H. Daumé III, and H. Wallach. 2020. Language (technology) is power: A critical survey of "bias" in NLP. *ACL*.

Borges, J. L. 1964. The analytical language of john wilkins. In *Other inquisitions 1937–1952*. University of Texas Press. Trans. Ruth L. C. Simms.

Caliskan, A., J. J. Bryson, and A. Narayanan. 2017. Semantics derived automatically from language corpora contain human-like biases. *Science*, 356(6334):183–186.

Chinchor, N., L. Hirschman, and D. L. Lewis. 1993. Evaluating Message Understanding systems: An analysis of the third Message Understanding Conference. *Computational Linguistics*, 19(3):409–449.

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

Crawford, K. 2017. The trouble with bias. Keynote at NeurIPS.

Davidson, T., D. Bhattacharya, and I. Weber. 2019. Racial bias in hate speech and abusive language detection datasets. *Third Workshop on Abusive Language Online*.

Dias Oliva, T., D. Antonialli, and A. Gomes. 2021. Fighting hate speech, silencing drag queens? artificial intelligence in content moderation and risks to lgbtq voices online. *Sexuality & Culture*, 25:700–732.

Dixon, L., J. Li, J. Sorensen, N. Thain, and L. Vasserman. 2018. Measuring and mitigating unintended bias in text classification. *2018 AAAI/ACM Conference on AI, Ethics, and Society*.

Doumbouya, M. K. B., D. Jurafsky, and C. D. Manning. 2025. Tversky neural networks: Psychologically plausible deep learning with differentiable tversky similarity. ArXiv preprint.

Dror, R., G. Baumer, M. Bogomolov, and R. Reichart. 2017. Replicability analysis for natural language processing: Testing significance with multiple datasets. *TACL*, 5:471––486.

Dror, R., L. Peled-Cohen, S. Shlomov, and R. Reichart. 2020. *Statistical Significance Testing for Natural Language Processing*, volume 45 of *Synthesis Lectures on Human Language Technologies*. Morgan & Claypool.

Efron, B. and R. J. Tibshirani. 1993. *An introduction to the bootstrap*. CRC press.

Gillick, L. and S. J. Cox. 1989. Some statistical issues in the comparison of speech recognition algorithms. *ICASSP*.

Guyon, I. and A. Elisseeff. 2003. An introduction to variable and feature selection. *JMLR*, 3:1157–1182.

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

Hutchinson, B., V. Prabhakaran, E. Denton, K. Webster, Y. Zhong, and S. Denuyl. 2020. Social biases in NLP models as barriers for persons with disabilities. *ACL*.

Jaech, A., G. Mulcaire, S. Hathi, M. Ostendorf, and N. A. Smith. 2016. Hierarchical character-word models for language identification. *ACL Workshop on NLP for Social Media*.

Jauhiainen, T., M. Lui, M. Zampieri, T. Baldwin, and K. Lindén. 2019. Automatic language identification in texts: A survey. *JAIR*, 65(1):675–682.

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

Kiritchenko, S. and S. M. Mohammad. 2018. Examining gender and race bias in two hundred sentiment analysis systems. **SEM*.

Liu, B. and L. Zhang. 2012. A survey of opinion mining and sentiment analysis. In C. C. Aggarwal and C. Zhai, eds, *Mining text data*, 415–464. Springer.

Manning, C. D., P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval*. Cambridge.

Mitchell, M., S. Wu, A. Zaldivar, P. Barnes, L. Vasserman, B. Hutchinson, E. Spitzer, I. D. Raji, and T. Gebru. 2019. Model cards for model reporting. *ACM FAccT*.

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

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

Noreen, E. W. 1989. *Computer Intensive Methods for Testing Hypothesis*. Wiley.

Pang, B. and L. Lee. 2008. Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, 2(1-2):1–135.

Park, J. H., J. Shin, and P. Fung. 2018. Reducing gender bias in abusive language detection. *EMNLP*.

Popp, D., R. A. Donovan, M. Crawford, K. L. Marsh, and M. Peele. 2003. Gender, race, and speech style stereotypes. *Sex Roles*, 48(7-8):317–325.

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.

Sap, M., D. Card, S. Gabriel, Y. Choi, and N. A. Smith. 2019. The risk of racial bias in hate speech detection. *ACL*.

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

Søgaard, A., A. Johannsen, B. Plank, D. Hovy, and H. M. Alonso. 2014. What's in a p-value in NLP? *CoNLL*.

Stamatatos, E. 2009. A survey of modern authorship attribution methods. *JASIST*, 60(3):538–556.

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

van Rijsbergen, C. J. 1975. *Information Retrieval*. Butterworths.

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

Yang, Y. and J. Pedersen. 1997. A comparative study on feature selection in text categorization. *ICML*.