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

observation

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 from Y. Sometimes you'll see the output classes referred to as the set C instead of Y.

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.

supervised machine learning

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

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):.

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

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 learning to find features in these training sentences (perhaps words like "awesome" or "awful"). **Probabilistic classifiers** like logistic regression are classifiers that in addition to giving an answer (the class this observation is in), also give the *probability* of the observation being in the class. This distribution over the classes can be useful information for downstream decisions; avoiding making discrete decisions early on can be useful when combining systems. Probabilistic classifiers like logistic regression have 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 the estimated class, by computing the probability $P(y = y_i|x)$ for each output class y_i. 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 probabilistic 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 $P(y = y_i|x)$ for each output class y_i. Then, given this vector of probabilities, we 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 4.7.

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

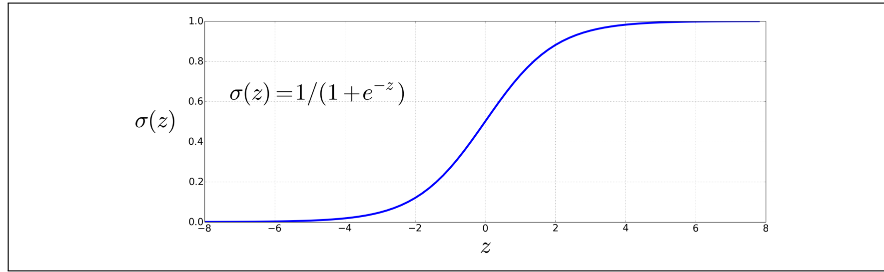


Figure 4.1 The sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $(0,1)$. It is nearly linear around 0 but outlier values get squashed toward 0 or 1.

$(0,1)$, 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, two terminological points. First, 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.

Second, for binary classification it will be convenient to refer to the output of the sigmoid function $\sigma(\mathbf{w}\cdot\mathbf{x}+b)$ which is computing $P(y=1)$, as $\hat{y}$. We pronounce this symbol as **y hat**. We thus use $\hat{y}$ to mean "the probability of the input observation having the positive class". When we introduce multinomial or softmax logistic regression in Section 4.7, we will introduce an extension of $\hat{y}$ that will be a vector of probabities over all the output classes, but for binary classification we just keep one scalar probability, knowing that we can always compute the missing probability $P(y=0)$ as $1-P(y=1)$.

y hat
$\hat{y}$

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 $x_1 \ldots x_6$ 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 $\in$ doc)	3
x_2	count(negative lexicon words $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	$\ln$(word+punctuation count of doc)	$\ln(66) = 4.19$

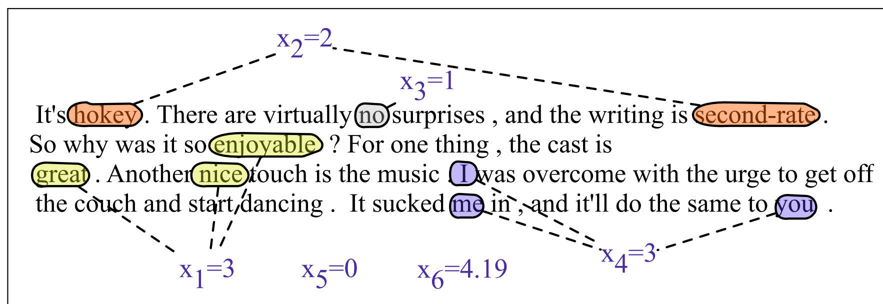


Figure 4.2 A sample mini test document showing the extracted features in the vector x.

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.

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

feature interactions

feature templates

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.

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** input values by centering them to result in a zero mean and a standard deviation of one (this transformation is sometimes called the **z-score**). That is, if μ_i is the mean of the values of feature x_i across the m observations in the input dataset, and σ_i is the standard deviation of the values of features x_i across the input dataset, we can replace each feature x_i by a new feature x'_i computed as follows:

standardize

z-score

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

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

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.

Another very common type of scaling for natural language data is to take logs, for example when inputs are word counts, or bigram counts, or anything else that follows a Zipfian distribution.

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 64, in which a superscript value in parentheses refers to the example index in some set of data (either for training or for test). So in this case each test example $x^{(i)}$ has a feature vector $\mathbf{x}^{(i)}$, $1 \leq i \leq m$. (As usual, we'll represent vectors and matrices in bold.)

One way to compute $\hat{y}^{(i)}$ (which is how we refer to $P(y^{(1)} = 1)$, i.e., the probability that the output for input $x^{(i)}i$ has the value 1), is just to have a for-loop and compute each test example one at a time:

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

For the first 3 test examples, then, we would be separately computing the predicted output probability as follows:

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

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 $\hat{y}^{(i)} = P(y^{(i)} = 1)$ 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 $\hat{\mathbf{y}}$, will correctly be:

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

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

$$\underset{(m\times 1)}{\hat{\mathbf{y}}} = \sigma(\underset{(m\times f)}{\mathbf{X}}\ \underset{(f\times 1)}{\mathbf{w}} + \underset{(m\times 1)}{\mathbf{b}}) \tag{4.15}$$

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 Learning in Logistic Regression

How are the parameters of the model, the weights $\mathbf{w}$ and bias b, learned? Binary 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 probability of y being 1, which we can think of an estimate of the true y (which is either 1 or 0). We want to learn parameters (meaning $\mathbf{w}$ and b) that make the probability value $\hat{y}$ for each training observation as close as possible

to the true y. That is, if y is 1, then we want the system's estimate $\hat{y} = P(y = 1)$ to be high, i.e. as close to 1 as possible. If y is in fact 0, then we want the system's estimate $\hat{y} = P(y = 1)$ to be low, i.e. as close to 0 as possible.

This requires two components that we foreshadowed in the introduction to the chapter. The first is a metric for how close $\hat{y}$ (the systems's estimate of the probability that the observation is in the positive class) is to the true gold label y. Rather than measure similarity, we usually talk about the opposite of this: the *distance* between the system output and the gold output, and we call this distance the **loss** function or the **cost function**. In the next section we'll introduce the loss function that is commonly used for logistic regression and also for neural networks, the **cross-entropy loss**.

loss

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

We'll describe these algorithms for the simpler case of binary logistic regression in the next two sections, and then turn to multinomial logistic regression in Section 4.8.

4.5 The cross-entropy loss function

We need a loss function that expresses, for an observation x, how close the classifier output ($\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$) is to the correct output (y, which is 0 or 1). Recall that $\hat{y}$ expresses the probability that the classifier assigns to observation x being in the positive class 1. So both $\hat{y}$ and y express probabilities, but y is always 0 or 1 but $\hat{y}$ can also lie in between. We'll call this measure of loss or distance:

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y \tag{4.16}$$

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. 4.17 simplifies to $\hat{y}$; if $y = 0$, Eq. 4.17 simplifies to $1 - \hat{y}$):

$$p(y|x) = \hat{y}^{y} \, (1 - \hat{y})^{1-y} \tag{4.17}$$

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:

$$\begin{aligned} \log p(y|x) &= \log \left[\hat{y}^{y} \, (1 - \hat{y})^{1-y} \right] \\ &= y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \end{aligned} \tag{4.18}$$

Eq. 4.18 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. 4.18. The result is the cross-entropy loss L_{CE}:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) \;=\; -[y \log \hat{y} + (1-y)\log(1-\hat{y})] \tag{4.19}$$

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

$$L_{\text{CE}}(\hat{y}, y) \;=\; -[y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \tag{4.20}$$

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. 4.20, 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. 4.20, 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 predicting the correct label (.36) is less than the loss for predicting the incorrect label (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. 4.18 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.

4.6 Gradient Descent

Our goal with gradient descent is to find the optimal weights: minimize the loss function we've defined for the model. In Eq. 4.21 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 we'll represent $\hat{y}^{(i)}$ as $f(x^{(i)};\theta)$ to make the dependence on θ more obvious. The goal is to find the set of weights which minimizes the loss function, averaged over all examples:

$$\hat{\theta} = \operatorname*{argmin}_{\theta} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)};\theta), y^{(i)}) \tag{4.21}$$

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. 4.3.

Given a random initialization of w at some value w^1, and assuming the loss function L happened to have the shape in Fig. 4.3, we need the algorithm to tell us whether at the next iteration we should move left (making w^2 smaller than w^1) or right (making w^2 bigger than w^1) to reach the minimum.

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. 4.3, we can informally think of the gradient as the slope. The dotted line in Fig. 4.3 shows the slope of this hypothetical loss function at point $w = w^1$. You can see that the slope of this dotted line is negative. Thus to find the minimum, gradient descent tells us to go in the opposite direction: moving w in a positive direction.

learning rate

The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{dw}L(f(x;w),y)$ 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):

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x;w),y) \tag{4.22}$$

Now let's extend the intuition from a function of one scalar variable w to many

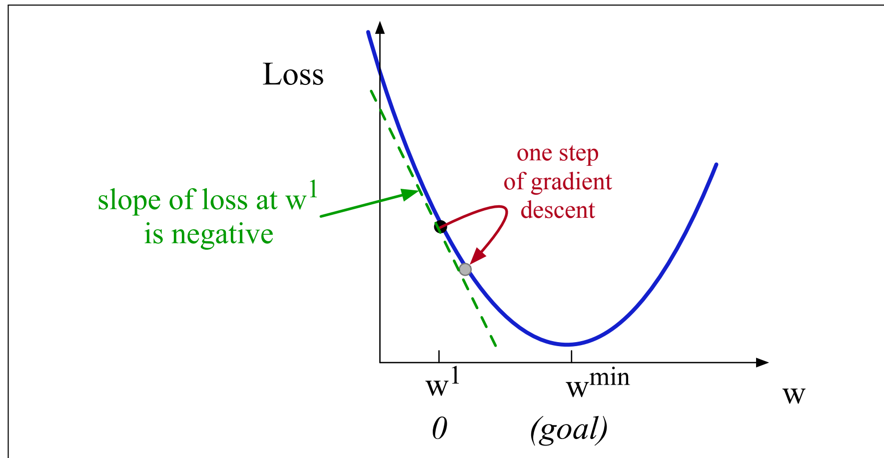


Figure 4.3 The first step in iteratively finding the minimum of this loss function, by moving w in the reverse direction from the slope of the function. Since the slope is negative, we need to move w in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of w (which is 0), w^2 the value at the second step, and so on.

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. 4.4 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?"

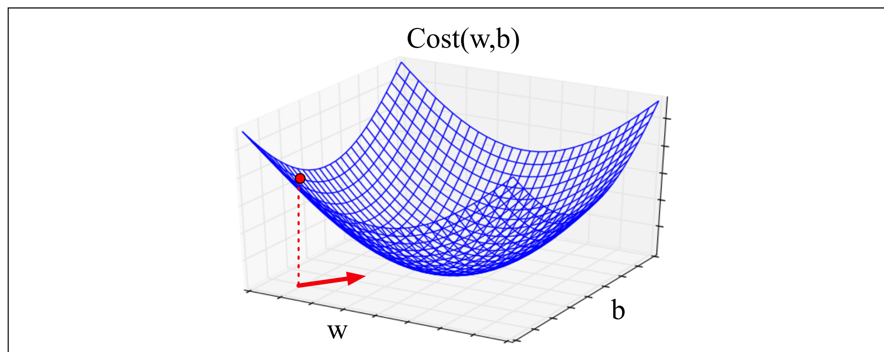


Figure 4.4 Visualization of the gradient vector at the red point in two dimensions w and b, showing a red arrow in the x-y plane pointing in the direction we will go to look for the minimum: the opposite direction of the gradient (recall that the gradient points in the direction of increase not decrease).

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:

$$\nabla L(f(x;\theta),y) = \begin{bmatrix} \frac{\partial}{\partial w_1}L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2}L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n}L(f(x;\theta),y) \\ \frac{\partial}{\partial b}L(f(x;\theta),y) \end{bmatrix} \tag{4.23}$$

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

$$\theta^{t+1} = \theta^t - \eta\nabla L(f(x;\theta),y) \tag{4.24}$$

4.6.1 The Gradient for Logistic Regression

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

$$L_{\text{CE}}(\hat{y},y) = -[y\log\sigma(\mathbf{w}\cdot\mathbf{x}+b)+(1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \tag{4.25}$$

It turns out that the derivative of this function for one observation vector x is Eq. 4.26 (the interested reader can see Section 4.15 for the derivation of this equation):

$$\begin{aligned} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j \\ &= (\hat{y}-y)x_j \end{aligned} \tag{4.26}$$

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

$$\frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial w_j} = -(y-\hat{y})x_j \tag{4.27}$$

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.

By the way, we'll also need a term for the partial derivative with respect to b. That turns out to be:

$$\begin{aligned} \frac{\partial L_{\text{CE}}(\hat{y},y)}{\partial b} &= \sigma(\mathbf{w}\cdot\mathbf{x}+b)-y \\ &= \hat{y}-y \end{aligned} \tag{4.28}$$

4.6.2 The Stochastic Gradient Descent Algorithm

Stochastic gradient descent is an online algorithm that minimizes the loss function by computing its gradient after each training example, and nudging θ in the right direction (the opposite direction of the gradient). (An "online algorithm" is one that processes its input example by example, rather than waiting until it sees the entire input.) Stochastic gradient descent is called **stochastic** because it chooses a single

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

θ ← 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 ŷ^(i) = f(x^(i); θ)          # What is our estimated output ŷ?
        Compute the loss L(ŷ^(i), y^(i))     # How far off is ŷ^(i) from the true output y^(i)?
     2. g ← ∇_θ L(f(x^(i); θ), y^(i))        # How should we move θ to maximize loss?
     3. θ ← θ − η g                          # Go the other way instead
return θ
```

Figure 4.5 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.

random example at a time; in Section 4.6.4 we'll discuss other versions of gradient descent that batch many examples at once. Fig. 4.5 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.

4.6.3 Working through an example

Let's walk through a single step of the gradient descent algorithm. We'll use a simplified version of the example in Fig. 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:

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

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

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

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

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

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

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

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

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

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

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

4.6.4 Mini-batch training

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

batch training

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

mini-batch

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

We just need to define mini-batch versions of the cross-entropy loss function we defined in Section 4.5 and the gradient in Section 4.6.1. Let's extend the cross-entropy loss for one example from Eq. 4.19 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:

$$\begin{aligned}
\log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}) \\
&= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}) \\
&= -\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \qquad (4.29)
\end{aligned}$$

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

$$\begin{aligned}
Cost(\hat{y}, y) &= \frac{1}{m}\sum_{i=1}^{m} L_{\text{CE}}(\hat{y}^{(i)}, y^{(i)}) \\
&= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b) + (1-y^{(i)}) \log\left(1-\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b)\right) \quad (4.30)
\end{aligned}$$

The mini-batch gradient is the average of the individual gradients from Eq. 4.26:

$$\frac{\partial Cost(\hat{y}, y)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left[\sigma(\mathbf{w}\cdot\mathbf{x}^{(i)}+b) - y^{(i)}\right] x_j^{(i)} \qquad (4.31)$$

Instead of using the sum notation, we can more efficiently compute the gradient in its matrix form, following the vectorization we saw on page 70, 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:

$$\begin{aligned}
\frac{\partial Cost(\hat{y}, y)}{\partial \mathbf{w}} &= \frac{1}{m}(\hat{\mathbf{y}} - \mathbf{y})^\intercal \mathbf{X} \\
&= \frac{1}{m}(\sigma(\mathbf{X}\mathbf{w}+\mathbf{b}) - \mathbf{y})^\intercal \mathbf{X} \qquad (4.32)
\end{aligned}$$

4.7 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 $\mathbf{y}$ for each input $\mathbf{x}$ will be a vector of length K. If class c is the correct class, we'll set $y_c = 1$, and set all the other elements of $\mathbf{y}$ to be 0, i.e., $y_c = 1$ and $y_j = 0 \quad \forall j \neq c$. A vector like this $\mathbf{y}$, with one value=1 and the rest 0, is called a **one-hot vector**. The job of the classifier is to produce an estimate vector $\hat{\mathbf{y}}$. For each class k, the value $\hat{y}_k$ will be the classifier's estimate of the probability $P(y_k = 1|\mathbf{x})$.

4.7.1 Softmax

softmax

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

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

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K}\exp(z_j)} \quad 1 \leq i \leq K \tag{4.33}$$

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

$$\text{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{K}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K}\exp(z_i)}, ..., \frac{\exp(z_K)}{\sum_{i=1}^{K}\exp(z_i)}\right] \tag{4.34}$$

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

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

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

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

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

Finally, note that, just as for the sigmoid, we refer to $\mathbf{z}$, the vector of scores that is the input to the softmax, as **logits** (see Eq. 4.7).

4.7.2 Applying softmax in logistic regression

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

$$P(y_k = 1|\mathbf{x}) = \frac{\exp(\mathbf{w}_k \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w}_j \cdot \mathbf{x} + b_j)} \tag{4.35}$$

The form of Eq. 4.35 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:

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad (4.36)$$

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 a protoype 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. 4.6 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.

4.7.3 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 67:

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

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.

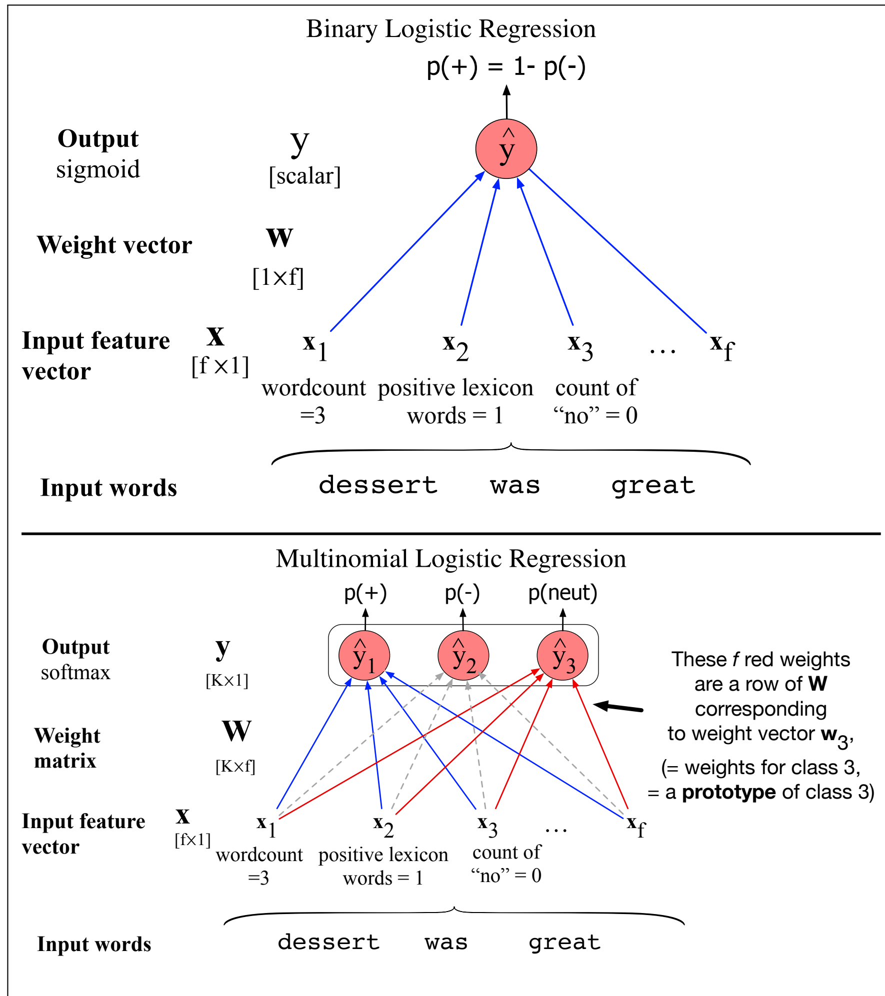


Figure 4.6 Binary versus multinomial logistic regression. Binary logistic regression uses a single weight vector $\mathbf{w}$, and has a scalar output $\hat{y}$. In multinomial logistic regression we have K separate weight vectors corresponding to the K classes, all packed into a single weight matrix $\mathbf{W}$, and a vector output $\hat{\mathbf{y}}$. We omit the biases from both figures for clarity.

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 the cross-entropy loss for binary logistic regression (repeated from Eq. 4.19) is:

$$L_{\text{CE}}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y} + (1-y)\log(1-\hat{y})] \tag{4.37}$$

The loss function for multinomial logistic regression generalizes the two terms in Eq. 4.37 (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. 4.38). This turns out to be just the negative log probability of the correct class c (Eq. 4.39):

$$L_{\text{CE}}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k \tag{4.38}$$

$$= -\log \hat{y}_c, \quad \text{(where } c \text{ is the correct class)} \tag{4.39}$$

$$= -\log \hat{p}(y_c = 1|\mathbf{x}) \quad \text{(where } c \text{ is the correct class)}$$

$$= -\log \frac{\exp(\mathbf{w_c} \cdot \mathbf{x} + b_c)}{\sum_{j=1}^{K} \exp(\mathbf{w_j} \cdot \mathbf{x} + b_j)} \quad (c \text{ is the correct class}) \tag{4.40}$$

How did we get from Eq. 4.38 to Eq. 4.39? 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. 4.38 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. 4.39 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. 4.26. 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:

$$\frac{\partial L_{\text{CE}}}{\partial w_{k,i}} = -(y_k - \hat{y}_k)x_i$$

$$= -(y_k - p(y_k = 1|x))x_i$$

$$= -\left(y_k - \frac{\exp(\mathbf{w_k} \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w_j} \cdot \mathbf{x} + b_j)}\right)x_i \tag{4.41}$$

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. 6.35–Eq. 6.43.

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.

confusion matrix

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.

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).

		gold standard labels		
		gold positive	gold negative	
system output labels	system positive	**true positive**	**false positive**	**precision** = $\frac{\text{tp}}{\text{tp+fp}}$
	system negative	**false negative**	**true negative**	
		recall = $\frac{\text{tp}}{\text{tp+fn}}$		**accuracy** = $\frac{\text{tp+tn}}{\text{tp+fp+tn+fn}}$

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

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 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:

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

F1

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:

$$\mathrm{F}_1 = \frac{2PR}{P+R} \tag{4.42}$$

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:

$$\text{HarmonicMean}(\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3, \mathrm{a}_4, ..., \mathrm{a}_\mathrm{n}) = \frac{\mathrm{n}}{\frac{1}{\mathrm{a}_1} + \frac{1}{\mathrm{a}_2} + \frac{1}{\mathrm{a}_3} + ... + \frac{1}{\mathrm{a}_\mathrm{n}}} \tag{4.43}$$

and hence F-measure is

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P+R} \tag{4.44}$$

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

The simple example we gave above in defining precision and recall involved text classification with only two classes. But as we saw in Section 4.7, lots of NLP classification tasks have more than two classes. Even for sentiment analysis we often have 3 classes (positive, negative, neutral) and many more classes are common in tasks like text classification or emotion detection, and so on.

To deal with more than two classes 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.

	gold labels urgent	*gold labels* normal	*gold labels* spam	
system output urgent	8	10	1	**precision**u= $\frac{8}{8+10+1}$
system output normal	5	60	50	**precision**n= $\frac{60}{5+60+50}$
system output 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.

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.

Class 1: Urgent

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

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

Class 2: Normal

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

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

Class 3: Spam

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

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

Pooled

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

microaverage precision = $\frac{268}{268+99}$ = **.73**

macroaverage precision = $\frac{.42+.52+.86}{3}$ = **.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.

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.

4.10 Test sets and Cross-validation

The training and testing procedure for text classification follows what we saw with language modeling (Section 3.2): 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.

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.

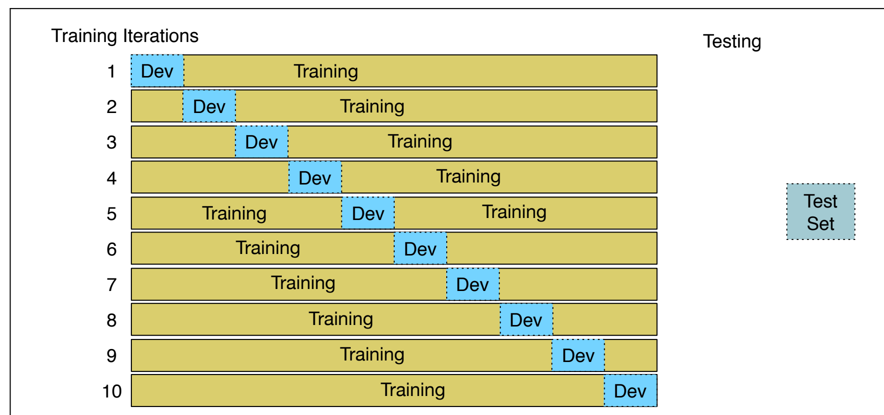


Figure 4.10 10-fold cross-validation

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:

$$\delta(x) = M(A,x) - M(B,x) \tag{4.45}$$

effect size

We would like to know if $\delta(x) > 0$, meaning that our new classifier A has a higher F_1 than our old classifier B 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.

$$\begin{aligned} H_0 &: \delta(x) \leq 0 \\ H_1 &: \delta(x) > 0 \end{aligned} \tag{4.46}$$

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.

p-value

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

$$P(\delta(X) \geq \delta(x) | H_0 \text{ is true}) \tag{4.47}$$

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.

statistically significant

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.

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.

approximate randomization

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

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.

4.11.1 The Paired Bootstrap Test

bootstrap test

bootstrapping

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.

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 (B̸) 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 (AB̸). 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 AB̸ 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.

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

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

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:

$$\begin{aligned}\text{p-value}(x) &= \frac{1}{b}\sum_{i=1}^{b}\mathbb{1}\left(\delta(x^{(i)}) - \delta(x) \geq \delta(x)\right) \\ &= \frac{1}{b}\sum_{i=1}^{b}\mathbb{1}\left(\delta(x^{(i)}) \geq 2\delta(x)\right) \qquad (4.48)\end{aligned}$$

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.

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

representational harms

It is important to avoid harms that may result from classifiers. 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

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 δ is probably not accidental

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

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 envi-

ronmental situations

4.13 Interpreting models

interpretable

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

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

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. 4.21, resulting in the following loss for a batch of m examples (slightly

rewritten from Eq. 4.21 to be maximizing log probability rather than minimizing loss, and removing the $\frac{1}{m}$ term which doesn't affect the argmax):

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha R(\theta) \quad (4.49)$$

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. Note that regularization is not normally applied to the bias term, which acts as a kind of threshold that helps deal with uncentered data and class priors.

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:

$$R(\theta) = ||\theta||_2^2 = \sum_{j=1}^{n} \theta_j^2 \quad (4.50)$$

The L2 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)};\theta) \right] - \alpha \sum_{j=1}^{n} \theta_j^2 \quad (4.51)$$

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):

$$R(\theta) = ||\theta||_1 = \sum_{i=1}^{n} |\theta_i| \quad (4.52)$$

The L1 regularized loss function becomes:

$$\hat{\theta} = \underset{\theta}{\text{argmax}} \left[\sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)};\theta) \right] - \alpha \sum_{j=1}^{n} |\theta_j| \quad (4.53)$$

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. Again, for both these types of regularization we general ignore the bias term and only consider the other weights.

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

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2}\right) \tag{4.54}$$

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) \times \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(\theta_j - \mu_j)^2}{2\sigma_j^2}\right) \tag{4.55}$$

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) - \alpha \sum_{j=1}^{n} \theta_j^2 \tag{4.56}$$

which is in the same form as Eq. 4.51.

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 $\ln(x)$:

$$\frac{d}{dx}\ln(x) = \frac{1}{x} \tag{4.57}$$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z)) \tag{4.58}$$

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:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx} \tag{4.59}$$

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):

$$
\begin{aligned}
\frac{\partial L_{\text{CE}}}{\partial w_j} &= \frac{\partial}{\partial w_j} - [y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))] \\
&= -\left[\frac{\partial}{\partial w_j} y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + \frac{\partial}{\partial w_j}(1-y)\log[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\right]
\end{aligned}
\tag{4.60}
$$

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

$$
\frac{\partial L_{\text{CE}}}{\partial w_j} = -\frac{y}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\cdot\mathbf{x}+b) - \frac{1-y}{1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}\frac{\partial}{\partial w_j}[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]
\tag{4.61}
$$

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. 4.62:

$$
\begin{aligned}
\frac{\partial L_{\text{CE}}}{\partial w_j} &= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]\frac{\partial(\mathbf{w}\cdot\mathbf{x}+b)}{\partial w_j} \\
&= -\left[\frac{y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)}{\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]}\right]\sigma(\mathbf{w}\cdot\mathbf{x}+b)[1-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j \\
&= -[y-\sigma(\mathbf{w}\cdot\mathbf{x}+b)]x_j \\
&= [\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y]x_j
\end{aligned}
\tag{4.62}
$$

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, 1997b), 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

CHAPTER

5 Embeddings

荃者所以在鱼，得鱼而忘荃	Nets are for fish; Once you get the fish, you can forget the net.
言者所以在意，得意而忘言	Words are for meaning; Once you get the meaning, you can forget the words

庄子(Zhuangzi), Chapter 26

The asphalt that Los Angeles is famous for occurs mainly on its freeways. But in the middle of the city is another patch of asphalt, the La Brea tar pits, and this asphalt preserves millions of fossil bones from the last of the Ice Ages of the Pleistocene Epoch. One of these fossils is the *Smilodon*, or saber-toothed tiger, instantly recognizable by its long canines. Five million years ago or so, a completely different saber-tooth tiger called *Thylacosmilus* lived in Argentina and other parts of South America. Thylacosmilus was a marsupial whereas Smilodon was a placental mammal, but Thylacosmilus had the same long upper canines and, like Smilodon, had a protective bone flange on the lower jaw. The similarity of these two mammals is one of many examples of parallel or convergent evolution, in which particular contexts or environments lead to the evolution of very similar structures in different species (Gould, 1980).

The role of context is also important in the similarity of a less biological kind of organism: the word. Words that occur in *similar contexts* tend to have *similar meanings*. This link between similarity in how words are distributed and similarity in what they mean is called the **distributional hypothesis**. The hypothesis was first formulated in the 1950s by linguists like Joos (1950), Harris (1954), and Firth (1957), who noticed that words which are synonyms (like *oculist* and *eye-doctor*) tended to occur in the same environment (e.g., near words like *eye* or *examined*) with the amount of meaning difference between two words "corresponding roughly to the amount of difference in their environments" (Harris, 1954, p. 157).

distributional hypothesis

In this chapter we introduce **embeddings**, vector representations of the meaning of words that are learned directly from word distributions in texts. Embeddings lie at the heart of large language models and other modern applications. The **static embeddings** we introduce here underlie the more powerful dynamic or **contextualized embeddings** like **BERT** that we will see in Chapter 9 and Chapter 8.

embeddings

The linguistic field that studies embeddings and their meanings is called **vector semantics**. Embeddings are also the first example in this book of **representation learning**, automatically learning useful representations of the input text. Finding such **self-supervised** ways to learn representations of language, instead of creating representations by hand via **feature engineering**, is an important principle of modern NLP (Bengio et al., 2013).

vector semantics
representation learning

5.1 Lexical Semantics

Let's begin by introducing some basic principles of word meaning. How should we represent the meaning of a word? In the n-gram models of Chapter 3, and in classical NLP applications, our only representation of a word is as a string of letters, or an index in a vocabulary list. This representation is not that different from a tradition in philosophy, perhaps you've seen it in introductory logic classes, in which the meaning of words is represented by just spelling the word with small capital letters; representing the meaning of "dog" as DOG, and "cat" as CAT, or by using an apostrophe (DOG').

Representing the meaning of a word by capitalizing it is a pretty unsatisfactory model. You might have seen a version of a joke due originally to semanticist Barbara Partee (Carlson, 1977):

> Q: What's the meaning of life?
> A: LIFE'

Surely we can do better than this! After all, we'll want a model of word meaning to do all sorts of things for us. It should tell us that some words have similar meanings (*cat* is similar to *dog*), others are antonyms (*cold* is the opposite of *hot*), some have positive connotations (*happy*) while others have negative connotations (*sad*). It should represent the fact that the meanings of *buy*, *sell*, and *pay* offer differing perspectives on the same underlying purchasing event. (If I buy something from you, you've probably sold it to me, and I likely paid you.) More generally, a model of word meaning should allow us to draw inferences to address meaning-related tasks like question-answering or dialogue.

lexical semantics

In this section we summarize some of these desiderata, drawing on results in the linguistic study of word meaning, which is called **lexical semantics**; we'll return to and expand on this list in Appendix G and Chapter 21.

Lemmas and Senses Let's start by looking at how one word (we'll choose *mouse*) might be defined in a dictionary (simplified from the online dictionary WordNet):

```
mouse (N)
1.  any of numerous small rodents...
2.  a hand-operated device that controls a cursor...
```

lemma

citation form

Here the form *mouse* is the **lemma**, also called the **citation form**. The form *mouse* would also be the lemma for the word *mice*; dictionaries don't have separate definitions for inflected forms like *mice*. Similarly *sing* is the lemma for *sing*, *sang*, *sung*. In many languages the infinitive form is used as the lemma for the verb, so Spanish *dormir* "to sleep" is the lemma for *duermes* "you sleep". The specific forms *sung* or *carpets* or *sing* or *duermes* are called **wordforms**.

wordform

As the example above shows, each lemma can have multiple meanings; the lemma *mouse* can refer to the rodent or the cursor control device. We call each of these aspects of the meaning of *mouse* a **word sense**. The fact that lemmas can be **polysemous** (have multiple senses) can make interpretation difficult (is someone who searches for "mouse info" looking for a pet or a widget?). Chapter 9 and Appendix G will discuss the problem of polysemy, and introduce **word sense disambiguation**, the task of determining which sense of a word is being used in a particular context.

Synonymy One important component of word meaning is the relationship between word senses. For example when one word has a sense whose meaning is

identical to a sense of another word, or nearly identical, we say the two senses of those two words are **synonyms**. Synonyms include such pairs as

synonym

couch/sofa vomit/throw up filbert/hazelnut car/automobile

A more formal definition of synonymy (between words rather than senses) is that two words are synonymous if they are substitutable for one another in any sentence without changing the *truth conditions* of the sentence, the situations in which the sentence would be true.

While substitutions between some pairs of words like *car* / *automobile* or *water* / H_2O are truth preserving, the words are still not identical in meaning. Indeed, probably no two words are absolutely identical in meaning. One of the fundamental tenets of semantics, called the **principle of contrast** (Girard 1718, Bréal 1897, Clark 1987), states that a difference in linguistic form is always associated with some difference in meaning. For example, the word H_2O is used in scientific contexts and would be inappropriate in a hiking guide—*water* would be more appropriate— and this genre difference is part of the meaning of the word. In practice, the word *synonym* is therefore used to describe a relationship of approximate or rough synonymy.

principle of contrast

Word Similarity While words don't have many synonyms, most words do have lots of *similar* words. *Cat* is not a synonym of *dog*, but *cats* and *dogs* are certainly similar words. In moving from synonymy to similarity, it will be useful to shift from talking about relations between word senses (like synonymy) to relations between words (like similarity). Dealing with words avoids having to commit to a particular representation of word senses, which will turn out to simplify our task.

similarity

The notion of word **similarity** is very useful in larger semantic tasks. Knowing how similar two words are can help in computing how similar the meaning of two phrases or sentences are, a very important component of tasks like question answering, paraphrasing, and summarization. One way of getting values for word similarity is to ask humans to judge how similar one word is to another. A number of datasets have resulted from such experiments. For example the SimLex-999 dataset (Hill et al., 2015) gives values on a scale from 0 to 10, like the examples below, which range from near-synonyms (*vanish*, *disappear*) to pairs that scarcely seem to have anything in common (*hole*, *agreement*):

vanish	disappear	9.8
belief	impression	5.95
muscle	bone	3.65
modest	flexible	0.98
hole	agreement	0.3

Word Relatedness The meaning of two words can be related in ways other than similarity. One such class of connections is called word **relatedness** (Budanitsky and Hirst, 2006), also traditionally called word **association** in psychology.

relatedness

association

Consider the meanings of the words *coffee* and *cup*. Coffee is not similar to cup; they share practically no features (coffee is a plant or a beverage, while a cup is a manufactured object with a particular shape). But coffee and cup are clearly related; they are associated by co-participating in an everyday event (the event of drinking coffee out of a cup). Similarly *scalpel* and *surgeon* are not similar but are related eventively (a surgeon tends to make use of a scalpel).

One common kind of relatedness between words is if they belong to the same **semantic field**. A semantic field is a set of words which cover a particular semantic domain and bear structured relations with each other. For example, words might be

semantic field

related by being in the semantic field of hospitals (*surgeon*, *scalpel*, *nurse*, *anesthetic*, *hospital*), restaurants (*waiter*, *menu*, *plate*, *food*, *chef*), or houses (*door*, *roof*, *kitchen*, *family*, *bed*). Semantic fields are also related to **topic models**, like **Latent Dirichlet Allocation**, **LDA**, which apply unsupervised learning on large sets of texts to induce sets of associated words from text. Semantic fields and topic models are very useful tools for discovering topical structure in documents.

topic models

In Appendix G we'll introduce more relations between senses like **hypernymy** or **IS-A**, **antonymy** (opposites) and **meronymy** (part-whole relations).

connotations

Connotation Finally, words have *affective meanings* or **connotations**. The word *connotation* has different meanings in different fields, but here we use it to mean the aspects of a word's meaning that are related to a writer or reader's emotions, sentiment, opinions, or evaluations. For example some words have positive connotations (*wonderful*) while others have negative connotations (*dreary*). Even words whose meanings are similar in other ways can vary in connotation; consider the difference in connotations between *fake*, *knockoff*, *forgery*, on the one hand, and *copy*, *replica*, *reproduction* on the other, or *innocent* (positive connotation) and *naive* (negative connotation). Some words describe positive evaluation (*great*, *love*) and others negative evaluation (*terrible*, *hate*). Positive or negative evaluation language is called **sentiment**, as we saw in Appendix K, and word sentiment plays a role in important tasks like sentiment analysis, stance detection, and applications of NLP to the language of politics and consumer reviews.

sentiment

Early work on affective meaning (Osgood et al., 1957) found that words varied along three important dimensions of affective meaning:

valence: the pleasantness of the stimulus
arousal: the intensity of emotion provoked by the stimulus
dominance: the degree of control exerted by the stimulus

Thus words like *happy* or *satisfied* are high on valence, while *unhappy* or *annoyed* are low on valence. *Excited* is high on arousal, while *calm* is low on arousal. *Controlling* is high on dominance, while *awed* or *influenced* are low on dominance. Each word is thus represented by three numbers, corresponding to its value on each of the three dimensions:

	Valence	Arousal	Dominance
courageous	8.0	5.5	7.4
music	7.7	5.6	6.5
heartbreak	2.5	5.7	3.6
cub	6.7	4.0	4.2

Osgood et al. (1957) noticed that in using these 3 numbers to represent the meaning of a word, the model was representing each word as a point in a three-dimensional space, a vector whose three dimensions corresponded to the word's rating on the three scales. This revolutionary idea that word meaning could be represented as a point in space (e.g., that part of the meaning of *heartbreak* can be represented as the point $[2.5, 5.7, 3.6]$) was the first expression of the vector semantics models that we introduce next.

5.2 Vector Semantics: The Intuition

vector semantics

Vector semantics is the standard way to represent word meaning in NLP, helping

us model many of the aspects of word meaning we saw in the previous section. The roots of the model lie in the 1950s when two big ideas converged: Osgood's 1957 idea mentioned above to use a point in three-dimensional space to represent the connotation of a word, and the proposal by linguists like Joos (1950), Harris (1954), and Firth (1957) to define the meaning of a word by its **distribution** in language use, meaning its neighboring words or grammatical environments. Their idea was that two words that occur in very similar distributions (whose neighboring words are similar) have similar meanings.

For example, suppose you didn't know the meaning of the word *ongchoi* (a recent borrowing from Cantonese) but you see it in the following contexts:

(5.1) Ongchoi is delicious sauteed with garlic.

(5.2) Ongchoi is superb over rice.

(5.3) ...ongchoi leaves with salty sauces...

And suppose that you had seen many of these context words in other contexts:

(5.4) ...spinach sauteed with garlic over rice...

(5.5) ...chard stems and leaves are delicious...

(5.6) ...collard greens and other salty leafy greens

The fact that *ongchoi* occurs with words like *rice* and *garlic* and *delicious* and *salty*, as do words like *spinach*, *chard*, and *collard greens* might suggest that ongchoi is a leafy green similar to these other leafy greens.[1] We can implement the same intuition computationally by just counting words in the context of *ongchoi*.

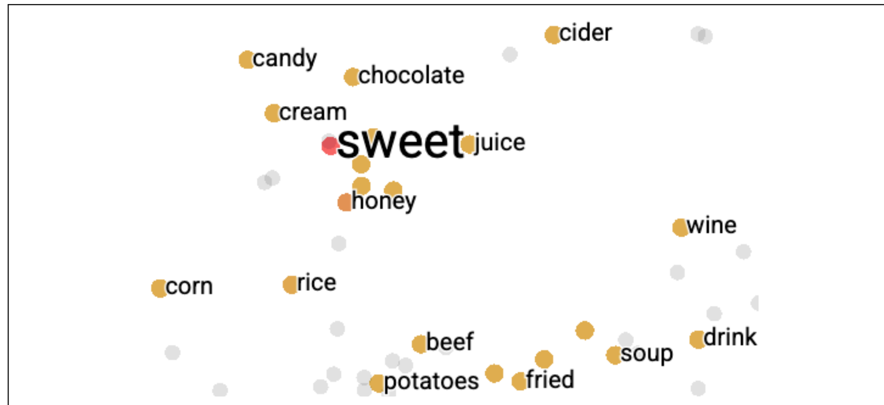


Figure 5.1 A two-dimensional (t-SNE) visualization of 200-dimensional word2vec embeddings for some words close to the word *sweet*, showing that words with similar meanings are nearby in space. Visualization created using the TensorBoard Embedding Projector `https://projector.tensorflow.org/`.

The idea of vector semantics is to represent a word as a point in a multidimensional semantic space that is derived (in different ways we'll see) from the distributions of word neighbors. Vectors for representing words are called **embeddings**. The word "embedding" derives historically from its mathematical sense as a mapping from one space or structure to another, although the meaning has shifted; see the end of the chapter.

embeddings

Fig. 5.1 shows a visualization of embeddings learned by the word2vec algorithm, showing the location of selected words (neighbors of "sweet") projected down from

[1] It's in fact *Ipomoea aquatica*, a relative of morning glory sometimes called *water spinach* in English.

200-dimensional space into a 2-dimensional space. Note that the nearest neighbors of sweet are semantically related words like honey, candy, juice, chocolate. This idea that similar words are neighbors in high-dimensional space offers enormous power to language models and other NLP applications. For example the sentiment classifiers of Chapter 4 depend on the same words appearing in the training and test sets. But by representing words as embeddings, a classifier can assign sentiment as long as it sees some words with *similar meanings*. And as we'll see, vector semantic models like the ones showed in Fig. 5.1 can be learned automatically from text without supervision.

In this chapter we'll begin with a simple pedagogical model of embeddings in which the meaning of a word is defined by a vector with the counts of nearby words. We introduce this model as a helpful way to understand the concept of vectors and what it means for a vector to be a representation of word meaning, but more sophisticated variants like the **tf-idf model** we will introduce in Chapter 11 are important methods you should understand. We will see that this method results in very long vectors that are **sparse**, i.e. mostly zeros (since most words simply never occur in the context of others). We'll then introduce the **word2vec** model family for constructing short, **dense** vectors that have even more useful semantic properties.

We'll also introduce the **cosine**, the standard way to use embeddings to compute *semantic similarity*, between two words, two sentences, or two documents, an important tool in practical applications.

5.3 Simple count-based embeddings

"The most important attributes of a vector in 3-space are {Location, Location, Location}"
Randall Munroe, the hover from https://xkcd.com/2358/

Let's now introduce the first way to compute word vector embeddings. This simplest vector model of meaning is based on the **co-occurrence matrix**, a way of representing how often words co-occur. We'll define a particular kind of co-occurrence matrix, the **word-context matrix**, in which each row in the matrix represents a word in the vocabulary and each column represents how often each other word in the vocabulary appears nearby. This matrix is thus of dimensionality $|V| \times |V|$ and each cell records the number of times the row (target) word and the column (context) word co-occur nearby in some training corpus.

word-context matrix

What do we mean by 'nearby'? We could implement various methods, but let's start with a very simple one: a context window around the word, let's say of 4 words to the left and 4 words to the right. If we do that, each cell will represents the number of times (in some training corpus) the column word occurs in such a ± 4 word window around the row word.

Let's see how this works for 4 words: *cherry*, *strawberry*, *digital*, and *information*. For each word we took a single instance from a corpus, and we show the ± 4 word window from that instance:

is traditionally followed by	**cherry**	pie, a traditional dessert
often mixed, such as	**strawberry**	rhubarb pie. Apple pie
computer peripherals and personal	**digital**	assistants. These devices usually
a computer. This includes	**information**	available on the internet

If we then take every occurrence of each word in a large corpus and count the context words around it, we get a word-context co-occurrence matrix. The full word-

context co-occurrence matrix is very large, because for each word in the vocabulary (since $|V|$) we have to count how often it occurs with every other word in the vocabulary, hence dimensionality $|V| \times |V|$. Let's therefore instead sketch the process on a smaller scale. Imagine that we are going to look at only the 4 words, and only consider the following 3 context words: *a*, *computer*, and *pie*. Furthermore let's assume we only count occurrences in the mini-corpus above.

So before looking at Fig. 5.2, compute by hand the counts for these 3 context words for the four words *cherry*, *strawberry*, *digital*, and *information*.

	a	**computer**	**pie**
cherry	1	0	1
strawberry	0	0	2
digital	0	1	0
information	1	1	0

Figure 5.2 Co-occurrence vectors for four words with counts from the 4 windows above, showing just 3 of the potential context word dimensions. The vector for *cherry* is outlined in red. Note that a real vector would have vastly more dimensions and thus be even sparser.

Hopefully your count matches what is shown in Fig. 5.2, so that each cell represents the number of times a particular word (defined by the row) occurs in a particular context (defined by the word column).

Each row, then, is a vector representing a word. To review some basic linear algebra, a **vector** is, at heart, just a list or array of numbers. So *cherry* is represented as the list [1,0,1] (the first **row vector** in Fig. 5.2) and *information* is represented as the list [1,1,0] (the fourth row vector).

vector

A **vector space** is a collection of vectors, and is characterized by its **dimension**. Vectors in a 3-dimensional vector space have an element for each dimension of the space. We will loosely refer to a vector in a 3-dimensional space as a 3-dimensional vector, with one element along each dimension. In the example in Fig. 5.2, we've chosen to make the document vectors of dimension 3, just so they fit on the page; in real term-document matrices, the document vectors would have dimensionality $|V|$, the vocabulary size.

vector space

dimension

The ordering of the numbers in a vector space indicates the different dimensions on which documents vary. The third dimension for all these vectors corresponds to the number of times *pie* occurs in the context. The second dimension for all of them corresponds to the number of times the word *computer* occurs. Notice that the vectors for *information* and *digital* have the same value (1) for this "computer" dimension.

In reality, we don't compute word vectors on a single context window. Instead, we compute them over an entire corpus. Let's see what some real counts look like. Let's look at some vectors computed in this way. Fig. 5.3 shows a subset of the word-word co-occurrence matrix for these four words, where, again because it's impossible to visualize all $|V|$ possible context words on the page of this textbook, we show a subset of 6 of the dimensions, with counts computed from the Wikipedia corpus (Davies, 2015).

Note in Fig. 5.3 that the two words *cherry* and *strawberry* are more similar to each other (both *pie* and *sugar* tend to occur in their window) than they are to other words like *digital*; conversely, *digital* and *information* are more similar to each other than, say, to *strawberry*.

We can think of the vector for a document as a point in $|V|$-dimensional space; thus the documents in Fig. 5.3 are points in 3-dimensional space. Fig. 5.4 shows a spatial visualization.

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Figure 5.3 Co-occurrence vectors for four words in the Wikipedia corpus, showing six of the dimensions (hand-picked for pedagogical purposes). The vector for *digital* is outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser, i.e. would have zero values in most dimensions.

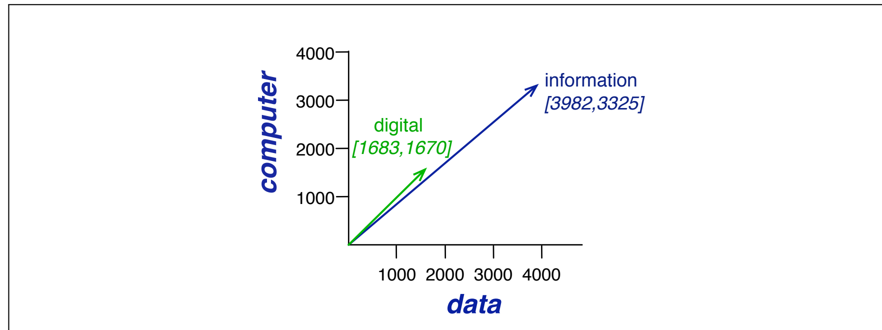


Figure 5.4 A spatial visualization of word vectors for *digital* and *information*, showing just two of the dimensions, corresponding to the words *data* and *computer*.

Note that $|V|$, the dimensionality of the vector, is generally the size of the vocabulary, often between 10,000 and 50,000 words (using the most frequent words in the training corpus; keeping words after about the most frequent 50,000 or so is generally not helpful). Since most of these numbers are zero these are **sparse** vector representations; there are efficient algorithms for storing and computing with sparse matrices.

It's also possible to apply various kinds of weighting functions to the counts in these cells. The most popular such weighting is tf-idf, which we'll introduce in Chapter 11, but there have historically been a wide variety of other weightings.

Now that we have some intuitions, let's move on to examine the details of computing word similarity.

5.4 Cosine for measuring similarity

To measure similarity between two target words v and w, we need a metric that takes two vectors (of the same dimensionality, either both with words as dimensions, hence of length $|V|$, or both with documents as dimensions, of length $|D|$) and gives a measure of their similarity. By far the most common similarity metric is the **cosine** of the angle between the vectors.

dot product
inner product

The cosine—like most measures for vector similarity used in NLP—is based on the **dot product** operator from linear algebra, also called the **inner product**:

$$\text{dot product}(\mathbf{v}, \mathbf{w}) = \mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N \tag{5.7}$$

The dot product acts as a similarity metric because it will tend to be high just when the two vectors have large values in the same dimensions. Alternatively, vectors that

have zeros in different dimensions—orthogonal vectors—will have a dot product of 0, representing their strong dissimilarity.

This raw dot product, however, has a problem as a similarity metric: it favors **long** vectors. The **vector length** is defined as

vector length

$$|\mathbf{v}| = \sqrt{\sum_{i=1}^{N} v_i^2} \tag{5.8}$$

The dot product is higher if a vector is longer, with higher values in each dimension. More frequent words have longer vectors, since they tend to co-occur with more words and have higher co-occurrence values with each of them. The raw dot product thus will be higher for frequent words. But this is a problem; we'd like a similarity metric that tells us how similar two words are regardless of their frequency.

We modify the dot product to normalize for the vector length by dividing the dot product by the lengths of each of the two vectors. This **normalized dot product** turns out to be the same as the cosine of the angle between the two vectors, following from the definition of the dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= |\mathbf{a}||\mathbf{b}| \cos\theta \\ \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} &= \cos\theta \end{aligned} \tag{5.9}$$

cosine

The **cosine** similarity metric between two vectors $\mathbf{v}$ and $\mathbf{w}$ thus can be computed as:

$$\text{cosine}(\mathbf{v}, \mathbf{w}) = \frac{\mathbf{v} \cdot \mathbf{w}}{|\mathbf{v}||\mathbf{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}} \tag{5.10}$$

unit vector

For some applications we pre-normalize each vector, by dividing it by its length, creating a **unit vector** of length 1. Thus we could compute a unit vector from $\mathbf{a}$ by dividing it by $|\mathbf{a}|$. For unit vectors, the dot product is the same as the cosine.

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for orthogonal vectors, to -1 for vectors pointing in opposite directions. But since raw frequency values are non-negative, the cosine for these vectors ranges from 0–1.

Let's see how the cosine computes which of the words *cherry* or *digital* is closer in meaning to *information*, just using raw counts from the following shortened table:

	pie	**data**	**computer**
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{cherry}, \text{information}) = \frac{442 * 5 + 8 * 3982 + 2 * 3325}{\sqrt{442^2 + 8^2 + 2^2}\sqrt{5^2 + 3982^2 + 3325^2}} = .018$$

$$\cos(\text{digital}, \text{information}) = \frac{5 * 5 + 1683 * 3982 + 1670 * 3325}{\sqrt{5^2 + 1683^2 + 1670^2}\sqrt{5^2 + 3982^2 + 3325^2}} = .996$$

The model decides that *information* is way closer to *digital* than it is to *cherry*, a result that seems sensible. Fig. 5.5 shows a visualization.

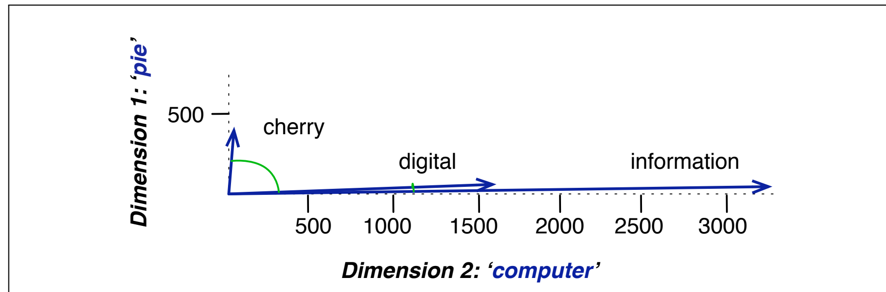


Figure 5.5 A (rough) graphical demonstration of cosine similarity, showing vectors for three words (*cherry*, *digital*, and *information*) in the two dimensional space defined by counts of the words *computer* and *pie* nearby. The figure doesn't show the cosine, but it highlights the angles; note that the angle between *digital* and *information* is smaller than the angle between *cherry* and *information*. When two vectors are more similar, the cosine is larger but the angle is smaller; the cosine has its maximum (1) when the angle between two vectors is smallest (0°); the cosine of all other angles is less than 1.

Cosine similarity can be used to estimate word similarity, for tasks like finding word paraphrases, tracking changes in word meaning, or automatically discovering meanings of words in different corpora. For example, we can fine the 10 most similar words to any target word w by computing the cosines between w and each of the $|V|-1$ other words, sorting, and looking at the top 10.

5.5 Word2vec

In the previous sections we saw how to represent a word as a sparse, long vector with dimensions corresponding to words in the vocabulary. We now introduce a more powerful word representation: **embeddings**, short dense vectors. Unlike the vectors we've seen so far, embeddings are **short**, with number of dimensions d ranging from 50-1000, rather than the much larger vocabulary size $|V|$.These d dimensions don't have a clear interpretation. And the vectors are **dense**: instead of vector entries being sparse, mostly-zero counts or functions of counts, the values will be real-valued numbers that can be negative.

It turns out that dense vectors work better in every NLP task than sparse vectors. While we don't completely understand all the reasons for this, we have some intuitions. Representing words as 300-dimensional dense vectors requires our classifiers to learn far fewer weights than if we represented words as 50,000-dimensional vectors, and the smaller parameter space possibly helps with generalization and avoiding overfitting. Dense vectors may also do a better job of capturing synonymy. For example, in a sparse vector representation, dimensions for synonyms like *car* and *automobile* dimension are distinct and unrelated; sparse vectors may thus fail to capture the similarity between a word with *car* as a neighbor and a word with *automobile* as a neighbor.

skip-gram **SGNS** **word2vec**

In this section we introduce one method for computing embeddings: **skip-gram with negative sampling**, sometimes called **SGNS**. The skip-gram algorithm is one of two algorithms in a software package called **word2vec**, and so sometimes the algorithm is loosely referred to as word2vec (Mikolov et al. 2013a, Mikolov et al. 2013b). The word2vec methods are fast, efficient to train, and easily available online with code and pretrained embeddings. Word2vec embeddings are **static em-**

static embeddings

beddings, meaning that the method learns one fixed embedding for each word in the vocabulary. In Chapter 9 we'll introduce methods for learning dynamic **contextual embeddings** like the popular family of **BERT** representations, in which the vector for each word is different in different contexts.

The intuition of word2vec is that instead of counting how often each context word c occurs near, say, *apricot*, we'll instead train a classifier on a binary prediction task: "Is word c likely to show up near *apricot*?" We don't actually care about this prediction task; instead we'll take the learned classifier *weights* as the word embeddings.

The revolutionary intuition here is that we can just use running text as implicitly supervised training data for such a classifier; a word c that occurs near the target word *apricot* acts as gold 'correct answer' to the question "Is word c likely to show up near *apricot*?" This method, often called **self-supervision**, avoids the need for any sort of hand-labeled supervision signal. This idea was first proposed in the task of neural language modeling, when Bengio et al. (2003) and Collobert et al. (2011) showed that a neural language model (a neural network that learned to predict the next word from prior words) could just use the next word in running text as its supervision signal, and could be used to learn an embedding representation for each word as part of doing this prediction task.

self-supervision

We'll see how to do neural networks in the next chapter, but word2vec is a much simpler model than the neural network language model, in two ways. First, word2vec simplifies the task (making it binary classification instead of word prediction). Second, word2vec simplifies the architecture (training a logistic regression classifier instead of a multi-layer neural network with hidden layers that demand more sophisticated training algorithms). The intuition of skip-gram is:

1. Treat the target word and a neighboring context word as positive examples.
2. Randomly sample other words in the lexicon to get negative samples.
3. Use logistic regression to train a classifier to distinguish those two cases.
4. Use the learned weights as the embeddings.

5.5.1 The classifier

Let's start by thinking about the classification task, and then turn to how to train. Imagine a sentence like the following, with a target word *apricot*, and assume we're using a window of ± 2 context words:

```
... lemon,  a [tablespoon of apricot jam,     a] pinch ...
               c1          c2    w     c3       c4
```

Our goal is to train a classifier such that, given a tuple (w,c) of a target word w paired with a candidate context word c (for example (*apricot*, *jam*), or perhaps (*apricot*, *aardvark*)) it will return the probability that c is a real context word (true for *jam*, false for *aardvark*):

$$P(+|w,c) \tag{5.11}$$

The probability that word c is not a real context word for w is just 1 minus Eq. 5.11:

$$P(-|w,c) = 1 - P(+|w,c) \tag{5.12}$$

How does the classifier compute the probability P? The intuition of the skip-gram model is to base this probability on embedding similarity: a word is likely to occur near the target if its embedding vector is similar to the target embedding. To compute similarity between these dense embeddings, we rely on the intuition that two vectors are similar if they have a high **dot product** (after all, cosine is just a normalized dot product). In other words:

$$Similarity(w,c) \approx \mathbf{c} \cdot \mathbf{w} \tag{5.13}$$

The dot product $\mathbf{c} \cdot \mathbf{w}$ is not a probability, it's just a number ranging from $-\infty$ to ∞ (since the elements in word2vec embeddings can be negative, the dot product can be negative). To turn the dot product into a probability, we'll use the **logistic** or **sigmoid** function $\sigma(x)$, the fundamental core of logistic regression:

$$\sigma(x) = \frac{1}{1+\exp(-x)} \tag{5.14}$$

We model the probability that word c is a real context word for target word w as:

$$P(+|w,c) = \sigma(\mathbf{c} \cdot \mathbf{w}) = \frac{1}{1+\exp(-\mathbf{c} \cdot \mathbf{w})} \tag{5.15}$$

The sigmoid function returns a number between 0 and 1, but to make it a probability we'll also need the total probability of the two possible events (c is a context word, and c isn't a context word) to sum to 1. We thus estimate the probability that word c is not a real context word for w as:

$$\begin{aligned} P(-|w,c) &= 1 - P(+|w,c) \\ &= \sigma(-\mathbf{c} \cdot \mathbf{w}) = \frac{1}{1+\exp(\mathbf{c} \cdot \mathbf{w})} \end{aligned} \tag{5.16}$$

Equation 5.15 gives us the probability for one word, but there are many context words in the window. Skip-gram makes the simplifying assumption that all context words are independent, allowing us to just multiply their probabilities:

$$P(+|w,c_{1:L}) = \prod_{i=1}^{L} \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{5.17}$$

$$\log P(+|w,c_{1:L}) = \sum_{i=1}^{L} \log \sigma(\mathbf{c_i} \cdot \mathbf{w}) \tag{5.18}$$

In summary, skip-gram trains a probabilistic classifier that, given a test target word w and its context window of L words $c_{1:L}$, assigns a probability based on how similar this context window is to the target word. The probability is based on applying the logistic (sigmoid) function to the dot product of the embeddings of the target word with each context word. To compute this probability, we just need embeddings for each target word and context word in the vocabulary.

Fig. 5.6 shows the intuition of the parameters we'll need. Skip-gram actually stores two embeddings for each word, one for the word as a target, and one for the word considered as context. Thus the parameters we need to learn are two matrices $\mathbf{W}$ and $\mathbf{C}$, each containing an embedding for every one of the $|V|$ words in the vocabulary V.[2] Let's now turn to learning these embeddings (which is the real goal of training this classifier in the first place).

[2] In principle the target matrix and the context matrix could use different vocabularies, but we'll simplify by assuming one shared vocabulary V.

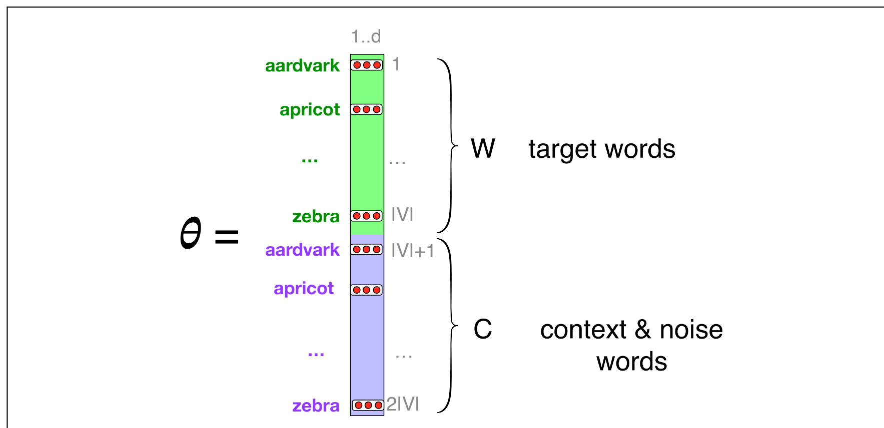


Figure 5.6 The embeddings learned by the skipgram model. The algorithm stores two embeddings for each word, the target embedding (sometimes called the input embedding) and the context embedding (sometimes called the output embedding). The parameter θ that the algorithm learns is thus a matrix of $2|V|$ vectors, each of dimension d, formed by concatenating two matrices, the target embeddings **W** and the context+noise embeddings **C**.

5.5.2 Learning skip-gram embeddings

The learning algorithm for skip-gram embeddings takes as input a corpus of text, and a chosen vocabulary size N. It begins by assigning a random embedding vector for each of the N vocabulary words, and then proceeds to iteratively shift the embedding of each word w to be more like the embeddings of words that occur nearby in texts, and less like the embeddings of words that don't occur nearby. Let's start by considering a single piece of training data:

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
               c1          c2    w     c3       c4
```

This example has a target word w (apricot), and 4 context words in the $L = \pm 2$ window, resulting in 4 positive training instances (on the left below):

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

For training a binary classifier we also need negative examples. In fact skip-gram with negative sampling (SGNS) uses more negative examples than positive examples (with the ratio between them set by a parameter k). So for each of these (w, c_{pos}) training instances we'll create k negative samples, each consisting of the target w plus a 'noise word' c_{neg}. A noise word is a random word from the lexicon, constrained not to be the target word w. The table right above shows the setting where $k = 2$, so we'll have 2 negative examples in the negative training set $-$ for each positive example w, c_{pos}.

The noise words are chosen according to their weighted unigram probability $p_\alpha(w)$, where α is a weight. If we were sampling according to unweighted probability $P(w)$, it would mean that with unigram probability $P(\text{"}the\text{"})$ we would choose the word *the* as a noise word, with unigram probability $P(\text{"}aardvark\text{"})$ we would

choose *aardvark*, and so on. But in practice it is common to set $\alpha = 0.75$, i.e. use the weighting $P_{\frac{3}{4}}(w)$:

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_{w'} count(w')^\alpha} \tag{5.19}$$

Setting $\alpha = .75$ gives better performance because it gives rare noise words slightly higher probability: for rare words, $P_\alpha(w) > P(w)$. To illustrate this intuition, it might help to work out the probabilities for an example with $\alpha = .75$ and two events, $P(a) = 0.99$ and $P(b) = 0.01$:

$$\begin{aligned} P_\alpha(a) &= \frac{.99^{.75}}{.99^{.75} + .01^{.75}} = 0.97 \\ P_\alpha(b) &= \frac{.01^{.75}}{.99^{.75} + .01^{.75}} = 0.03 \end{aligned} \tag{5.20}$$

Thus using $\alpha = .75$ increases the probability of the rare event b from 0.01 to 0.03.

Given the set of positive and negative training instances, and an initial set of embeddings, the goal of the learning algorithm is to adjust those embeddings to

- Maximize the similarity of the target word, context word pairs (w, c_{pos}) drawn from the positive examples
- Minimize the similarity of the (w, c_{neg}) pairs from the negative examples.

If we consider one word/context pair (w, c_{pos}) with its k noise words $c_{neg_1} \dots c_{neg_k}$, we can express these two goals as the following loss function L to be minimized (hence the $-$); here the first term expresses that we want the classifier to assign the real context word c_{pos} a high probability of being a neighbor, and the second term expresses that we want to assign each of the noise words c_{neg_i} a high probability of being a non-neighbor, all multiplied because we assume independence:

$$\begin{aligned} L(w, c_{pos}, c_{neg*}) &= -\log\left[P(+|w, c_{pos}) \prod_{i=1}^{k} P(-|w, c_{neg_i})\right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log P(-|w, c_{neg_i})\right] \\ &= -\left[\log P(+|w, c_{pos}) + \sum_{i=1}^{k} \log\left(1 - P(+|w, c_{neg_i})\right)\right] \\ &= -\left[\log \sigma(c_{pos} \cdot w) + \sum_{i=1}^{k} \log \sigma(-c_{neg_i} \cdot w)\right] \end{aligned} \tag{5.21}$$

That is, we want to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the k negative sampled non-neighbor words.

We minimize this loss function using stochastic gradient descent. Fig. 5.7 shows the intuition of one step of learning.

To get the gradient, we need to take the derivative of Eq. 5.21 with respect to the different embeddings. It turns out the derivatives are the following (we leave the

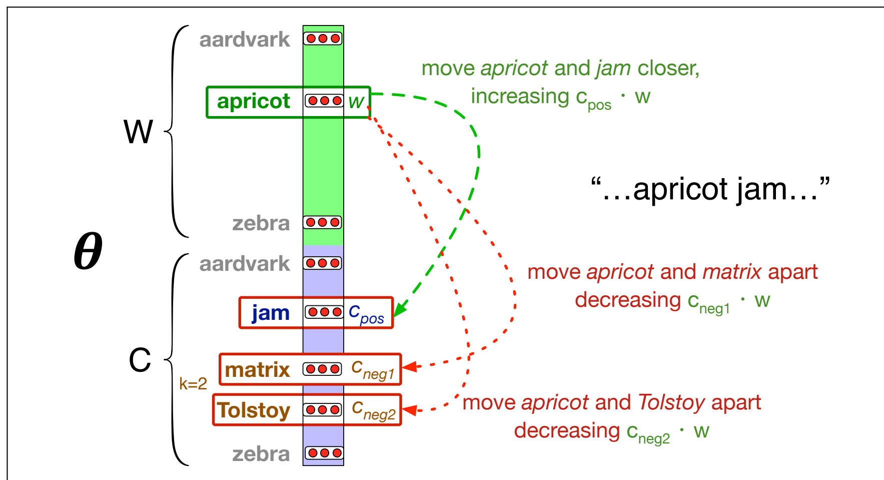


Figure 5.7 Intuition of one step of gradient descent. The skip-gram model tries to shift embeddings so the target embeddings (here for *apricot*) are closer to (have a higher dot product with) context embeddings for nearby words (here *jam*) and further from (lower dot product with) context embeddings for noise words that don't occur nearby (here *Tolstoy* and *matrix*).

proof as an exercise at the end of the chapter):

$$\frac{\partial L}{\partial c_{pos}} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{w} \tag{5.22}$$

$$\frac{\partial L}{\partial c_{neg_i}} = [\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w})]\mathbf{w} \tag{5.23}$$

$$\frac{\partial L}{\partial w} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{c}_{pos} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w})]\mathbf{c}_{neg_i} \tag{5.24}$$

The update equations going from time step t to $t+1$ in stochastic gradient descent are thus:

$$\mathbf{c}_{pos}^{t+1} = \mathbf{c}_{pos}^{t} - \eta[\sigma(\mathbf{c}_{pos}^{t} \cdot \mathbf{w}^{t}) - 1]\mathbf{w}^{t} \tag{5.25}$$

$$\mathbf{c}_{neg_i}^{t+1} = \mathbf{c}_{neg_i}^{t} - \eta[\sigma(\mathbf{c}_{neg_i}^{t} \cdot \mathbf{w}^{t})]\mathbf{w}^{t} \tag{5.26}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta\left[[\sigma(\mathbf{c}_{pos}^{t} \cdot \mathbf{w}^{t}) - 1]\mathbf{c}_{pos}^{t} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i}^{t} \cdot \mathbf{w}^{t})]\mathbf{c}_{neg_i}^{t}\right] \tag{5.27}$$

Just as in logistic regression, then, the learning algorithm starts with randomly initialized **W** and **C** matrices, and then walks through the training corpus using gradient descent to move **W** and **C** so as to minimize the loss in Eq. 5.21 by making the updates in (Eq. 5.25)-(Eq. 5.27).

Recall that the skip-gram model learns **two** separate embeddings for each word i: the **target embedding** $\mathbf{w}_i$ and the **context embedding** $\mathbf{c}_i$, stored in two matrices, the **target matrix W** and the **context matrix C**. It's common to just add them together, representing word i with the vector $\mathbf{w}_i + \mathbf{c}_i$. Alternatively we can throw away the **C** matrix and just represent each word i by the vector $\mathbf{w}_i$.

target embedding
context embedding

As with the simple count-based methods like tf-idf, the context window size affects the performance of skip-gram embeddings, and experiments often tune the context window size parameter on a devset.

5.5.3 Other kinds of static embeddings

fasttext

There are many kinds of static embeddings. An extension of word2vec, **fasttext** (Bojanowski et al., 2017), addresses a problem with word2vec as we have presented it so far: it has no good way to deal with **unknown words**—words that appear in a test corpus but were unseen in the training corpus. A related problem is word sparsity, such as in languages with rich morphology, where some of the many forms for each noun and verb may only occur rarely. Fasttext deals with these problems by using subword models, representing each word as itself plus a bag of constituent n-grams, with special boundary symbols $<$ and $>$ added to each word. For example, with $n = 3$ the word *where* would be represented by the sequence `<where>` plus the character n-grams:

```
<wh, whe, her, ere, re>
```

Then a skipgram embedding is learned for each constituent n-gram, and the word *where* is represented by the sum of all of the embeddings of its constituent n-grams. Unknown words can then be presented only by the sum of the constituent n-grams. A fasttext open-source library, including pretrained embeddings for 157 languages, is available at `https://fasttext.cc`.

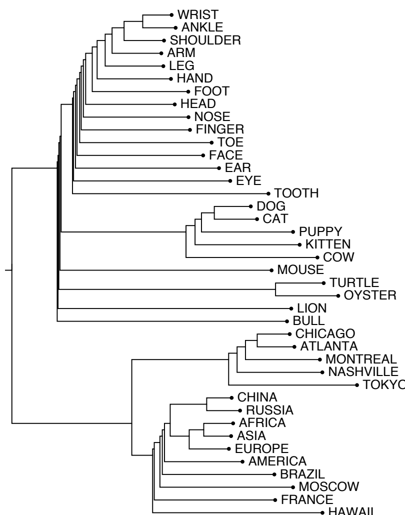
Another very widely used static embedding model is GloVe (Pennington et al., 2014), short for Global Vectors, because the model is based on capturing global corpus statistics. GloVe is based on ratios of probabilities from the word-word co-occurrence matrix.

It turns out that dense embeddings like word2vec actually have an elegant mathematical relationship with count-based embeddings, in which word2vec can be seen as implicitly optimizing a function of a count matrix with a particular (PPMI) weighting (Levy and Goldberg, 2014c).

5.6 Visualizing Embeddings

"I see well in many dimensions as long as the dimensions are around two."
The late economist Martin Shubik

Visualizing embeddings is an important goal in helping understand, apply, and improve these models of word meaning. But how can we visualize a (for example) 100-dimensional vector?



The simplest way to visualize the meaning of a word w embedded in a space is to list the most similar words to w by sorting the vectors for all words in the vocabulary by their cosine with the vector for w. For example the 7 closest words to *frog* using a particular set of embeddings computed with the GloVe algorithm are: *frogs*, *toad*, *litoria*, *leptodactylidae*, *rana*, *lizard*, and *eleutherodactylus* (Pennington et al., 2014).

Yet another visualization method is to use a clustering algorithm to show a hierarchical representation of which words are similar to others in the embedding space. The uncaptioned figure on the left uses hierarchical clustering of some embedding vectors for nouns as a visualization method (Rohde et al., 2006).

Probably the most common visualization method, however, is to project the 100 dimensions of a word down into 2 dimensions. Fig. 5.1 showed one such visualization, as does Fig. 5.9, using a projection method called t-SNE (van der Maaten and Hinton, 2008).

5.7 Semantic properties of embeddings

In this section we briefly summarize some of the semantic properties of embeddings that have been studied.

Different types of similarity or association: One parameter of vector semantic models that is relevant to both sparse PPMI vectors and dense word2vec vectors is the size of the context window used to collect counts. This is generally between 1 and 10 words on each side of the target word (for a total context of 2-20 words).

The choice depends on the goals of the representation. Shorter context windows tend to lead to representations that are a bit more syntactic, since the information is coming from immediately nearby words. When the vectors are computed from short context windows, the most similar words to a target word w tend to be semantically similar words with the same parts of speech. When vectors are computed from long context windows, the highest cosine words to a target word w tend to be words that are topically related but not similar.

For example Levy and Goldberg (2014a) showed that using skip-gram with a window of ± 2, the most similar words to the word *Hogwarts* (from the *Harry Potter* series) were names of other fictional schools: *Sunnydale* (from *Buffy the Vampire Slayer*) or *Evernight* (from a vampire series). With a window of ± 5, the most similar words to *Hogwarts* were other words topically related to the *Harry Potter* series: *Dumbledore*, *Malfoy*, and *half-blood*.

first-order co-occurrence

second-order co-occurrence

It's also often useful to distinguish two kinds of similarity or association between words (Schütze and Pedersen, 1993). Two words have **first-order co-occurrence** (sometimes called **syntagmatic association**) if they are typically nearby each other. Thus *wrote* is a first-order associate of *book* or *poem*. Two words have **second-order co-occurrence** (sometimes called **paradigmatic association**) if they have similar neighbors. Thus *wrote* is a second-order associate of words like *said* or *remarked*.

parallelogram model

Analogy/Relational Similarity: Another semantic property of embeddings is their ability to capture relational meanings. In an important early vector space model of cognition, Rumelhart and Abrahamson (1973) proposed the **parallelogram model** for solving simple analogy problems of the form *a is to b as a* is to what?*. In such problems, a system is given a problem like *apple:tree::grape:?*, i.e., *apple is to tree as grape is to* ____, and must fill in the word *vine*. In the parallelogram model, illustrated in Fig. 5.8, the vector from the word *apple* to the word *tree* (= $\overrightarrow{\text{tree}} - \overrightarrow{\text{apple}}$) is added to the vector for *grape* ($\overrightarrow{\text{grape}}$); the nearest word to that point is returned.

In early work with sparse embeddings, scholars showed that sparse vector models of meaning could solve such analogy problems (Turney and Littman, 2005), but the parallelogram method received more modern attention because of its success with word2vec or GloVe vectors (Mikolov et al. 2013c, Levy and Goldberg 2014b, Pennington et al. 2014). For example, the result of the expression $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is a vector close to $\overrightarrow{\text{queen}}$. Similarly, $\overrightarrow{\text{Paris}} - \overrightarrow{\text{France}} + \overrightarrow{\text{Italy}}$ results in a vector that is close to $\overrightarrow{\text{Rome}}$. The embedding model thus seems to be extract-

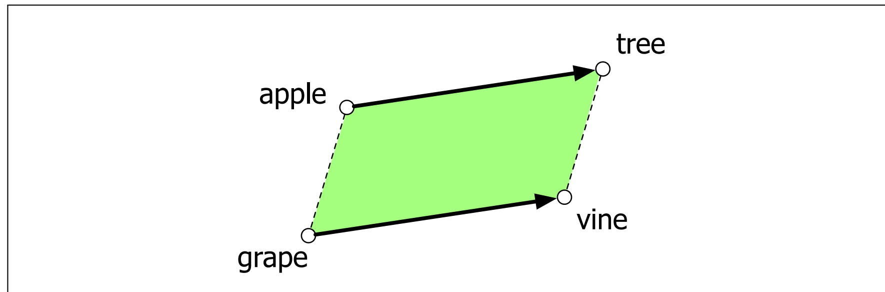


Figure 5.8 The parallelogram model for analogy problems (Rumelhart and Abrahamson, 1973): the location of $\overrightarrow{\text{vine}}$ can be found by subtracting $\overrightarrow{\text{apple}}$ from $\overrightarrow{\text{tree}}$ and adding $\overrightarrow{\text{grape}}$.

ing representations of relations like MALE-FEMALE, or CAPITAL-CITY-OF, or even COMPARATIVE/SUPERLATIVE, as shown in Fig. 5.9 from GloVe.

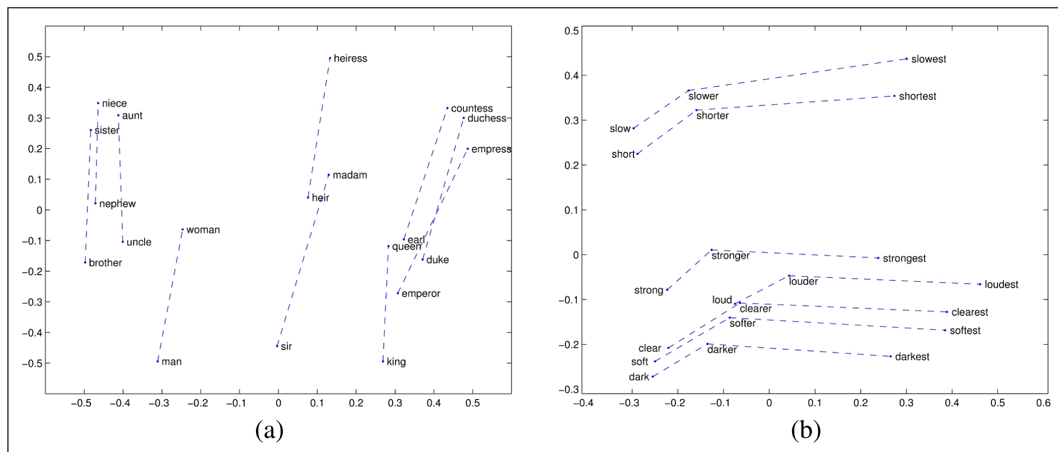


Figure 5.9 Relational properties of the GloVe vector space, shown by projecting vectors onto two dimensions. (a) $\overrightarrow{\text{king}} - \overrightarrow{\text{man}} + \overrightarrow{\text{woman}}$ is close to $\overrightarrow{\text{queen}}$. (b) offsets seem to capture comparative and superlative morphology (Pennington et al., 2014).

For a $\mathbf{a} : \mathbf{b} :: \mathbf{a}^* : \mathbf{b}^*$ problem, meaning the algorithm is given vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{a}^*$ and must find $\mathbf{b}^*$, the parallelogram method is thus:

$$\hat{\mathbf{b}}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \text{ distance}(\mathbf{x}, \mathbf{b} - \mathbf{a} + \mathbf{a}^*) \tag{5.28}$$

with some distance function, such as Euclidean distance.

There are some caveats. For example, the closest value returned by the parallelogram algorithm in word2vec or GloVe embedding spaces is usually not in fact b* but one of the 3 input words or their morphological variants (i.e., *cherry:red :: potato:x* returns *potato* or *potatoes* instead of *brown*), so these must be explicitly excluded. Furthermore while embedding spaces perform well if the task involves frequent words, small distances, and certain relations (like relating countries with their capitals or verbs/nouns with their inflected forms), the parallelogram method with embeddings doesn't work as well for other relations (Linzen 2016, Gladkova et al. 2016, Schluter 2018, Ethayarajh et al. 2019a), and indeed Peterson et al. (2020) argue that the parallelogram method is in general too simple to model the human cognitive process of forming analogies of this kind.

5.7.1 Embeddings and Historical Semantics

Embeddings can also be a useful tool for studying how meaning changes over time, by computing multiple embedding spaces, each from texts written in a particular time period. For example Fig. 5.10 shows a visualization of changes in meaning in English words over the last two centuries, computed by building separate embedding spaces for each decade from historical corpora like Google n-grams (Lin et al., 2012b) and the Corpus of Historical American English (Davies, 2012).

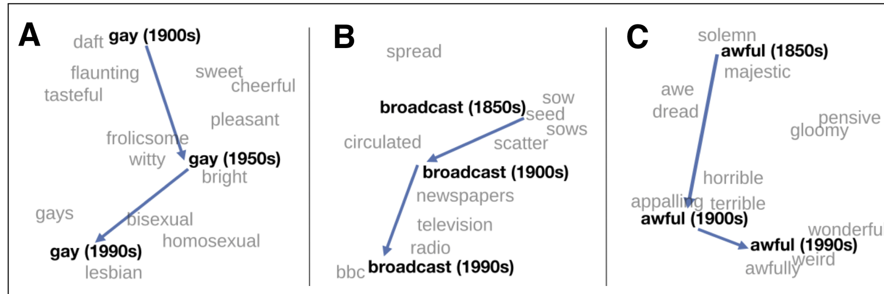


Figure 5.10 A t-SNE visualization of the semantic change of 3 words in English using word2vec vectors. The modern sense of each word, and the grey context words, are computed from the most recent (modern) time-point embedding space. Earlier points are computed from earlier historical embedding spaces. The visualizations show the changes in the word *gay* from meanings related to "cheerful" or "frolicsome" to referring to homosexuality, the development of the modern "transmission" sense of *broadcast* from its original sense of sowing seeds, and the pejoration of the word *awful* as it shifted from meaning "full of awe" to meaning "terrible or appalling" (Hamilton et al., 2016b).

5.8 Bias and Embeddings

In addition to their ability to learn word meaning from text, embeddings, alas, also reproduce the implicit biases and stereotypes that were latent in the text. As the prior section just showed, embeddings can roughly model relational similarity: 'queen' as the closest word to 'king' - 'man' + 'woman' implies the analogy *man:woman::king:queen*. But these same embedding analogies also exhibit gender stereotypes. For example Bolukbasi et al. (2016) find that the closest occupation to 'computer programmer' - 'man' + 'woman' in word2vec embeddings trained on news text is 'homemaker', and that the embeddings similarly suggest the analogy 'father' is to 'doctor' as 'mother' is to 'nurse'. This could result in what Crawford (2017) and Blodgett et al. (2020) call an **allocational harm**, when a system allocates resources (jobs or credit) unfairly to different groups. For example algorithms that use embeddings as part of a search for hiring potential programmers or doctors might thus incorrectly downweight documents with women's names.

allocational harm

It turns out that embeddings don't just reflect the statistics of their input, but also **amplify** bias; gendered terms become **more** gendered in embedding space than they were in the input text statistics (Zhao et al. 2017, Ethayarajh et al. 2019b, Jia et al. 2020), and biases are more exaggerated than in actual labor employment statistics (Garg et al., 2018).

bias amplification

Embeddings also encode the implicit associations that are a property of human reasoning. The Implicit Association Test (Greenwald et al., 1998) measures peo-

ple's associations between concepts (like 'flowers' or 'insects') and attributes (like 'pleasantness' and 'unpleasantness') by measuring differences in the latency with which they label words in the various categories.[3] Using such methods, people in the United States have been shown to associate African-American names with unpleasant words (more than European-American names), male names more with mathematics and female names with the arts, and old people's names with unpleasant words (Greenwald et al. 1998, Nosek et al. 2002a, Nosek et al. 2002b). Caliskan et al. (2017) replicated all these findings of implicit associations using GloVe vectors and cosine similarity instead of human latencies. For example African-American names like 'Leroy' and 'Shaniqua' had a higher GloVe cosine with unpleasant words while European-American names ('Brad', 'Greg', 'Courtney') had a higher cosine with pleasant words. These problems with embeddings are an example of a **representational harm** (Crawford 2017, Blodgett et al. 2020), which is a harm caused by a system demeaning or even ignoring some social groups. Any embedding-aware algorithm that made use of word sentiment could thus exacerbate bias against African Americans.

representational harm

Recent research focuses on ways to try to remove these kinds of biases, for example by developing a transformation of the embedding space that removes gender stereotypes but preserves definitional gender (Bolukbasi et al. 2016, Zhao et al. 2017) or changing the training procedure (Zhao et al., 2018b). However, although these sorts of **debiasing** may reduce bias in embeddings, they do not eliminate it (Gonen and Goldberg, 2019), and this remains an open problem.

debiasing

Historical embeddings are also being used to measure biases in the past. Garg et al. (2018) used embeddings from historical texts to measure the association between embeddings for occupations and embeddings for names of various ethnicities or genders (for example the relative cosine similarity of women's names versus men's to occupation words like 'librarian' or 'carpenter') across the 20th century. They found that the cosines correlate with the empirical historical percentages of women or ethnic groups in those occupations. Historical embeddings also replicated old surveys of ethnic stereotypes; the tendency of experimental participants in 1933 to associate adjectives like 'industrious' or 'superstitious' with, e.g., Chinese ethnicity, correlates with the cosine between Chinese last names and those adjectives using embeddings trained on 1930s text. They also were able to document historical gender biases, such as the fact that embeddings for adjectives related to competence ('smart', 'wise', 'thoughtful', 'resourceful') had a higher cosine with male than female words, and showed that this bias has been slowly decreasing since 1960. We return in later chapters to this question about the role of bias in natural language processing.

5.9 Evaluating Vector Models

The most important evaluation metric for vector models is extrinsic evaluation on tasks, i.e., using vectors in an NLP task and seeing whether this improves performance over some other model.

[3] Roughly speaking, if humans associate 'flowers' with 'pleasantness' and 'insects' with 'unpleasantness', when they are instructed to push a green button for 'flowers' (daisy, iris, lilac) and 'pleasant words' (love, laughter, pleasure) and a red button for 'insects' (flea, spider, mosquito) and 'unpleasant words' (abuse, hatred, ugly) they are faster than in an incongruous condition where they push a red button for 'flowers' and 'unpleasant words' and a green button for 'insects' and 'pleasant words'.

Nonetheless it is useful to have intrinsic evaluations. The most common metric is to test their performance on **similarity**, computing the correlation between an algorithm's word similarity scores and word similarity ratings assigned by humans. **WordSim-353** (Finkelstein et al., 2002) is a commonly used set of ratings from 0 to 10 for 353 noun pairs; for example (*plane*, *car*) had an average score of 5.77. **SimLex-999** (Hill et al., 2015) is a more complex dataset that quantifies similarity (*cup, mug*) rather than relatedness (*cup, coffee*), and includes concrete and abstract adjective, noun and verb pairs. The **TOEFL dataset** is a set of 80 questions, each consisting of a target word with 4 additional word choices; the task is to choose which is the correct synonym, as in the example: *Levied is closest in meaning to: imposed, believed, requested, correlated* (Landauer and Dumais, 1997). All of these datasets present words without context.

Slightly more realistic are intrinsic similarity tasks that include context. The Stanford Contextual Word Similarity (SCWS) dataset (Huang et al., 2012) and the Word-in-Context (WiC) dataset (Pilehvar and Camacho-Collados, 2019) offer richer evaluation scenarios. SCWS gives human judgments on 2,003 pairs of words in their sentential context, while WiC gives target words in two sentential contexts that are either in the same or different senses; see Appendix G. The *semantic textual similarity* task (Agirre et al. 2012, Agirre et al. 2015) evaluates the performance of sentence-level similarity algorithms, consisting of a set of pairs of sentences, each pair with human-labeled similarity scores.

Another task used for evaluation is the analogy task, discussed on page 112, where the system has to solve problems of the form *a is to b as a* is to b**, given *a, b,* and *a** and having to find *b** (Turney and Littman, 2005). A number of sets of tuples have been created for this task (Mikolov et al. 2013a, Mikolov et al. 2013c, Gladkova et al. 2016), covering morphology (*city:cities::child:children*), lexicographic relations (*leg:table::spout:teapot*) and encyclopedia relations (*Beijing:China::Dublin:Ireland*), some drawing from the SemEval-2012 Task 2 dataset of 79 different relations (Jurgens et al., 2012).

All embedding algorithms suffer from inherent variability. For example because of randomness in the initialization and the random negative sampling, algorithms like word2vec may produce different results even from the same dataset, and individual documents in a collection may strongly impact the resulting embeddings (Tian et al. 2016, Hellrich and Hahn 2016, Antoniak and Mimno 2018). When embeddings are used to study word associations in particular corpora, therefore, it is best practice to train multiple embeddings with bootstrap sampling over documents and average the results (Antoniak and Mimno, 2018).

5.10 Summary

- In vector semantics, a word is modeled as a vector—a point in high-dimensional space, also called an **embedding**. In this chapter we focus on **static embeddings**, where each word is mapped to a fixed embedding.
- Vector semantic models fall into two classes: **sparse** and **dense**. In sparse models each dimension corresponds to a word in the vocabulary V and cells are functions of **co-occurrence counts**. The **word-context** or **term-term** matrix has a row for each (target) word in the vocabulary and a column for each context term in the vocabulary.

- Dense vector models typically have dimensionality 50–1000. **Word2vec** algorithms like **skip-gram** are a popular way to compute dense embeddings. Skip-gram trains a logistic regression classifier to compute the probability that two words are 'likely to occur nearby in text'. This probability is computed from the dot product between the embeddings for the two words.
- Skip-gram uses stochastic gradient descent to train the classifier, by learning embeddings that have a high dot product with embeddings of words that occur nearby and a low dot product with noise words.
- Other important embedding algorithms include **GloVe**, a method based on ratios of word co-occurrence probabilities.
- Whether using sparse or dense vectors, word and document similarities are computed by some function of the **dot product** between vectors. The cosine of two vectors—a normalized dot product—is the most popular such metric.

Historical Notes

The idea of vector semantics arose out of research in the 1950s in three distinct fields: linguistics, psychology, and computer science, each of which contributed a fundamental aspect of the model.

The idea that meaning is related to the distribution of words in context was widespread in linguistic theory of the 1950s, among distributionalists like Zellig Harris, Martin Joos, and J. R. Firth, and semioticians like Thomas Sebeok. As Joos (1950) put it,

> the linguist's "meaning" of a morpheme... is by definition the set of conditional probabilities of its occurrence in context with all other morphemes.

The idea that the meaning of a word might be modeled as a point in a multi-dimensional semantic space came from psychologists like Charles E. Osgood, who had been studying how people responded to the meaning of words by assigning values along scales like *happy/sad* or *hard/soft*. Osgood et al. (1957) proposed that the meaning of a word in general could be modeled as a point in a multidimensional Euclidean space, and that the similarity of meaning between two words could be modeled as the distance between these points in the space.

mechanical indexing

A final intellectual source in the 1950s and early 1960s was the field then called **mechanical indexing**, now known as **information retrieval**. In what became known as the **vector space model** for information retrieval (Salton 1971, Sparck Jones 1986), researchers demonstrated new ways to define the meaning of words in terms of vectors (Switzer, 1965), and refined methods for word similarity based on measures of statistical association between words like mutual information (Giuliano, 1965) and idf (Sparck Jones, 1972), and showed that the meaning of documents could be represented in the same vector spaces used for words. Around the same time, (Cordier, 1965) showed that factor analysis of word association probabilities could be used to form dense vector representations of words.

Some of the philosophical underpinning of the distributional way of thinking came from the late writings of the philosopher Wittgenstein, who was skeptical of the possibility of building a completely formal theory of meaning definitions for each word. Wittgenstein suggested instead that "the meaning of a word is its use in the language" (Wittgenstein, 1953, PI 43). That is, instead of using some logical language to define each word, or drawing on denotations or truth values, Wittgenstein's

idea is that we should define a word by how it is used by people in speaking and understanding in their day-to-day interactions, thus prefiguring the movement toward embodied and experiential models in linguistics and NLP (Glenberg and Robertson 2000, Lake and Murphy 2021, Bisk et al. 2020, Bender and Koller 2020).

More distantly related is the idea of defining words by a vector of discrete features, which has roots at least as far back as Descartes and Leibniz (Wierzbicka 1992, Wierzbicka 1996). By the middle of the 20th century, beginning with the work of Hjelmslev (Hjelmslev, 1969) (originally 1943) and fleshed out in early models of generative grammar (Katz and Fodor, 1963), the idea arose of representing meaning with **semantic features**, symbols that represent some sort of primitive meaning. For example words like *hen*, *rooster*, or *chick*, have something in common (they all describe chickens) and something different (their age and sex), representable as:

semantic feature

hen	`+female, +chicken, +adult`
rooster	`-female, +chicken, +adult`
chick	`+chicken, -adult`

The dimensions used by vector models of meaning to define words, however, are only abstractly related to this idea of a small fixed number of hand-built dimensions. Nonetheless, there has been some attempt to show that certain dimensions of embedding models do contribute some specific compositional aspect of meaning like these early semantic features.

The use of dense vectors to model word meaning, and indeed the term **embedding**, grew out of the **latent semantic indexing** (LSI) model (Deerwester et al., 1988) recast as **LSA** (**latent semantic analysis**) (Deerwester et al., 1990). In LSA **singular value decomposition—SVD—** is applied to a term-document matrix (each cell weighted by log frequency and normalized by entropy), and then the first 300 dimensions are used as the LSA embedding. Singular Value Decomposition (SVD) is a method for finding the most important dimensions of a dataset, those dimensions along which the data varies the most. LSA was then quickly widely applied: as a cognitive model (Landauer and Dumais, 1997), and for tasks like spell checking (Jones and Martin, 1997), language modeling (Bellegarda 1997, Coccaro and Jurafsky 1998, Bellegarda 2000), morphology induction (Schone and Jurafsky 2000, Schone and Jurafsky 2001b), multiword expressions (MWEs) (Schone and Jurafsky, 2001a), and essay grading (Rehder et al., 1998). Related models were simultaneously developed and applied to word sense disambiguation by Schütze (1992b). LSA also led to the earliest use of embeddings to represent words in a probabilistic classifier, in the logistic regression document router of Schütze et al. (1995). The idea of SVD on the term-term matrix (rather than the term-document matrix) as a model of meaning for NLP was proposed soon after LSA by Schütze (1992b). Schütze applied the low-rank (97-dimensional) embeddings produced by SVD to the task of word sense disambiguation, analyzed the resulting semantic space, and also suggested possible techniques like dropping high-order dimensions. See Schütze (1997).

SVD

A number of alternative matrix models followed on from the early SVD work, including Probabilistic Latent Semantic Indexing (PLSI) (Hofmann, 1999), Latent Dirichlet Allocation (LDA) (Blei et al., 2003), and Non-negative Matrix Factorization (NMF) (Lee and Seung, 1999).

The LSA community seems to have first used the word “embedding” in Landauer et al. (1997), in a variant of its mathematical meaning as a mapping from one space or mathematical structure to another. In LSA, the word embedding seems to have described the mapping from the space of sparse count vectors to the latent space of

SVD dense vectors. Although the word thus originally meant the mapping from one space to another, it has metonymically shifted to mean the resulting dense vector in the latent space, and it is in this sense that we currently use the word.

By the next decade, Bengio et al. (2003) and Bengio et al. (2006) showed that neural language models could also be used to develop embeddings as part of the task of word prediction. Collobert and Weston (2007), Collobert and Weston (2008), and Collobert et al. (2011) then demonstrated that embeddings could be used to represent word meanings for a number of NLP tasks. Turian et al. (2010) compared the value of different kinds of embeddings for different NLP tasks. Mikolov et al. (2011) showed that recurrent neural nets could be used as language models. The idea of simplifying the hidden layer of these neural net language models to create the skip-gram (and also CBOW) algorithms was proposed by Mikolov et al. (2013a). The negative sampling training algorithm was proposed in Mikolov et al. (2013b). There are numerous surveys of static embeddings and their parameterizations (Bullinaria and Levy 2007, Bullinaria and Levy 2012, Lapesa and Evert 2014, Kiela and Clark 2014, Levy et al. 2015).

See Manning et al. (2008) and Chapter 11 for a deeper understanding of the role of vectors in information retrieval, including how to compare queries with documents, more details on tf-idf, and issues of scaling to very large datasets. See Kim (2019) for a clear and comprehensive tutorial on word2vec. Cruse (2004) is a useful introductory linguistic text on lexical semantics.

Exercises