CHAPTER

7 Classification: Naive Bayes, Logistic Regression, Sentiment

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

Classification lies at the heart of both human and machine intelligence. Deciding what letter, word, or image has been presented to our senses, recognizing faces or voices, sorting mail, assigning grades to homeworks, are all examples of assigning a class or category to an input. The potential challenges of this task are highlighted by the fabulist Jorge Luis Borges (1964), who imagined classifying 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.*

While many language processing tasks can be productively viewed as tasks of classification, the classes are luckily far more practical than those of Borges. In this chapter we present two general algorithms for classification, demonstrated on one important set of classification problems: **text categorization**, the task of classifying an entire text by assigning it a label drawn from some set of labels.

text categorization

sentiment analysis

We focus on one common text categorization task, **sentiment analysis**, the extraction of **sentiment**, the positive or negative orientation that a writer expresses toward some object. A review of a movie, book, or product on the web expresses the author's sentiment toward the product, while an editorial or political text expresses sentiment toward a candidate or political action. Automatically extracting consumer sentiment is important for marketing of any sort of product, while measuring public sentiment is important for politics and also for market prediction. The simplest version of sentiment analysis is a binary classification task, and the words of the review provide excellent cues. Consider, for example, the following phrases extracted from positive and negative reviews of movies and restaurants,. Words like *great*, *richly*, *awesome*, and *pathetic*, and *awful* and *ridiculously* are very informative cues:

\+ *...zany characters and richly applied satire, and some great plot twists*
− *It was pathetic. The worst part about it was the boxing scenes...*
\+ *...awesome caramel sauce and sweet toasty almonds. I love this place!*
− *...awful pizza and ridiculously overpriced...*

spam detection

Spam detection is another important commercial application, the binary classification task of assigning an email to to one of the two classes *spam* or *not-spam*. Many lexical and other features can be used to perform this classification. For example you might quite reasonably be suspicious of an email containing phrases like "online pharmaceutical" or "WITHOUT ANY COST" or "Dear Winner".

authorship attribution

Another thing we might want to know about a text is its author. Determining a text's author, **authorship attribution**, and author characteristics like gender, age, and native language are text classification tasks that are relevant to the digital humanities, social sciences, and forensics as well as natural language processing.

Finally, one of the oldest tasks in text classification is assigning a library subject category or topic label to a text. Deciding whether a research paper concerns epidemiology or instead, perhaps, embryology, is an important component of information retrieval. Various sets of subject categories exist, such as the MeSH (Medical Subject Headings) thesaurus. In fact, as we will see, subject category classification is the task for which the naive Bayes algorithm was invented in 1961.

Classification is important far beyond the task of text classification. We've already seen other classification tasks: period disambiguation (deciding if a period is the end of a sentence or part of a word), word tokenization (deciding if a character should be a word boundary). Even language modeling can 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. In future chapters we will see that a part-of-speech tagger classifies each occurrence of a word in a sentence as, e.g., a noun or a verb, and a named-entity tagging system classifies whether a sequence of words refers to people, organizations, dates, or something else.

The goal of classification is to take a single observation, extract some useful features, and thereby **classify** the observation into one of a set of discrete classes. One method for classifying text is to use hand-written rules. There are many areas of language processing where hand-written rule-based classifiers constitute a state-of-the-art system, or at least part of it.

Rules can be fragile, however, as situations or data change over time, and for some tasks humans aren't necessarily good at coming up with the rules. Most cases of classification in language processing are therefore done via **supervised machine learning**, and this will be the subject of the remainder of this chapter.

Formally, the task of classification is to take an input x and a fixed set of output classes $Y = y_1, y_2, ..., y_M$ and return a predicted class $y \in Y$. For text classification, we'll sometimes talk about c (for "class") instead of y as our output variable, and d (for "document") instead of x as our input variable. In the supervised situation we have a training set of N document that have each been hand-labeled with a class: $(d_1, c_1), ..., (d_N, c_N)$. Our goal is to learn a classifier that is capable of mapping from a new document d to its correct class $c \in C$. A **probabilistic classifier** additionally will tell us the probability of the observation being in the class. This full distribution over the classes can be useful information for downstream decisions; avoiding making discrete decisions early on can be useful when combining systems.

Many kinds of machine learning algorithms are used to build classifiers. We will discuss two in depth in this chapter: multinomial naive Bayes and multinomial logistic regression, also known as the maximum entropy or MaxEnt classifier. These exemplify two ways of doing classification. **Generative** classifiers like naive Bayes build a model of each class. Given an observation, they return the class most likely to have generated the observation. **Discriminative** classifiers like logistic regression instead learn what features from the input are most useful to discriminate between the different possible classes. While discriminative systems are often more accurate and hence more commonly used, generative classifiers still have a role.

Other classifiers commonly used in language processing include support-vector machines (SVMs), random forests, perceptrons, and neural networks; see the end of the chapter for pointers.

7.1 Naive Bayes Classifiers

naive Bayes classifier

In this section we introduce the **multinomial naive Bayes classifier**, so called because it is a Bayesian classifier that makes a simplifying (naive) assumption about how the features interact.

bag-of-words

The intuition of the classifier is shown in Fig. 7.1. We represent a text document as if it were a **bag-of-words**, that is, an unordered set of words with their position ignored, keeping only their frequency in the document. In the example in the figure, instead of representing the word order in all the phrases like "I love this movie" and "I would recommend it", we simply note that the word *I* occurred 5 times in the entire excerpt, the word *it* 6 times, the word *love*, *recommend*, and *movie* once, and so on.

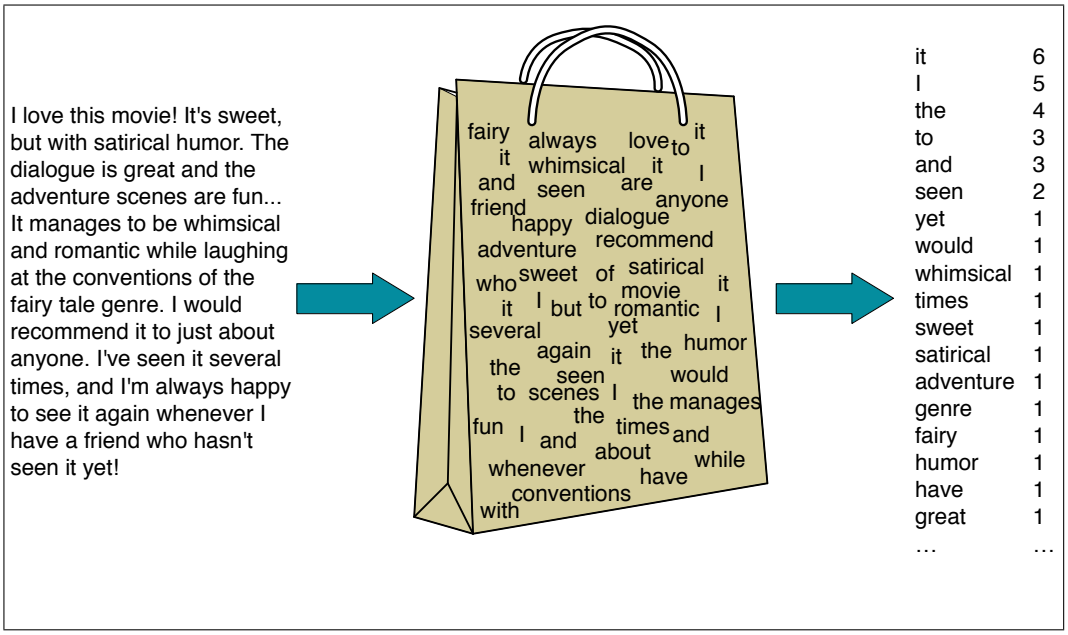

Figure 7.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

Naive Bayes is a probabilistic classifier, meaning that for a document d, out of all classes $c \in C$ the classifier returns the class $\hat{c}$ which has the maximum posterior probability given the document, In Eq. 7.1 we use the hat notation ˆ to mean "our estimate of the correct class".

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c|d) \tag{7.1}$$

Bayesian inference

This idea of **Bayesian inference** has been known since the work of Bayes (1763), and was first applied to text classification by Mosteller and Wallace (1964). The intuition of Bayesian classification is to use Bayes' rule to transform Eq. 7.1 into other probabilities that have some useful properties. Bayes' rule is presented in Eq. 7.2; it gives us a way to break down any conditional probability $P(x|y)$ into three other

probabilities:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \tag{7.2}$$

We can then substitute Eq. 7.2 into Eq. 7.1 to get Eq. 7.3:

$$\hat{c} = \operatorname*{argmax}_{c \in C} P(c|d) = \operatorname*{argmax}_{c \in C} \frac{P(d|c)P(c)}{P(d)} \tag{7.3}$$

We can conveniently simplify Eq. 7.3 by dropping the denominator $P(d)$. This is possible because we will be computing $\frac{P(d|c)P(c)}{P(d)}$ for each possible class. But $P(d)$ doesn't change for each class; we are always asking about the most likely class for the same document d, which must have the same probability $P(d)$. Thus, we can choose the class that maximizes this simpler formula:

$$\hat{c} = \operatorname*{argmax}_{c \in C} P(c|d) = \operatorname*{argmax}_{c \in C} P(d|c)P(c) \tag{7.4}$$

prior probability
likelihood

We thus compute the most probable class $\hat{c}$ given some document d by choosing the class which has the highest product of two probabilities: the **prior probability** of the class $P(t_1^n)$ and the **likelihood** of the document $P(d|c)$:

$$\hat{c} = \operatorname*{argmax}_{c \in C} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \tag{7.5}$$

Without loss of generalization, we can represent a document d as a set of features $f_1, f_2, ..., f_n$:

$$\hat{c} = \operatorname*{argmax}_{c \in C} \overbrace{P(f_1, f_2, ..., f_n|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \tag{7.6}$$

Unfortunately, Eq. 7.6 is still too hard to compute directly: without some simplifying assumptions, estimating the probability of every possible combination of feature (for example, every possible set of words and positions) would require huge numbers of parameters and impossibly large training sets. Naive Bayes classifiers therefore make two simplifying assumptions.

The first is the *bag of words* assumption discussed intuitively above: we assume position doesn't matter, and that the word "love" has the same effect on classification whether it occurs as the 1st, 20th, or last word in the document. Thus we assume that the features $f_1, f_2, ..., f_n$ only encode word identity and not position.

naive Bayes assumption

The second is commonly called the **naive Bayes assumption**: this is the conditional independence assumption that the probabilities $P(f_i|c)$ are independent given the class c and hence can be 'naively' multiplied as follows:

$$P(f_1, f_2, ..., f_n|c) = P(f_1|c) \cdot P(f_2|c) \cdot ... \cdot P(f_n|c) \tag{7.7}$$

The final equation for the class chosen by a naive Bayes classifier is thus:

$$c_{NB} = \operatorname*{argmax}_{c \in C} P(c) \prod_{f \in F} P(f|c) \tag{7.8}$$

To apply the naive Bayes classifier to text, we need to consider word positions, by simply walking an index through every word position in the document:

$$\begin{aligned} \text{positions} &\leftarrow \text{all word positions in test document} \\ c_{NB} &= \operatorname*{argmax}_{c \in C} P(c) \prod_{i \in positions} P(w_i|c) \end{aligned} \tag{7.9}$$

Naive Bayes calculations, like calculations for language modeling, are done in log space, to avoid underflow and increase speed. Thus Eq. 7.9 is generally instead expressed as

$$c_{NB} = \operatorname*{argmax}_{c \in C} \log P(c) + \sum_{i \in positions} \log P(w_i|c) \tag{7.10}$$

By considering features in log space Eq. 7.10 computes the predicted class as a linear function of input features. Classifier that use a linear combination of the inputs to make a classification decision —like naive Bayes and also logistic regression— are called **linear classifiers**.

linear classifiers

7.1.1 Training the Naive Bayes Classifier

How can we learn the probabilities $P(c)$ and $P(f_i|c)$? Let's first consider the maximum likelihood estimate. We'll simply use the frequencies in the data. For the document prior $P(c)$ we ask what percentage of the documents in our training set are in each class c. Let N_c be the number of documents in our training data with class c and N_{doc} be the total number of documents. Then:

$$\hat{P}(c) = \frac{N_c}{N_{doc}} \tag{7.11}$$

(7.12)

The learn the probability $P(f_i|c)$, we'll assume a feature is just the existence of a word in the document's bag of words, and so we'll want $P(w_i|c)$, which we compute as the fraction of times the word w_i appears among all words in all documents of topic c. We first concatenate all documents with category c into one big "category c" text. Then we use the frequency of w_i in this concatenated document to give a maximum likelihood estimate of the probability:

$$\hat{P}(w_i|c) = \frac{count(w_i, c)}{\sum_{w \in V} count(w, c)} \tag{7.13}$$

There is a problem, however, with maximum likelihood training. Imagine we are trying to estimate the likelihood of the word "fantastic" given class *positive*, but suppose there are no training documents that both contain the word "fantastic" and are classified as *positive*. Perhaps the word "fantastic" happens to occur (sarcastically?) in the class *negative*. In such a case the probability for this feature will be zero:

$$\hat{P}(\text{"fantastic"}|\text{positive}) = \frac{count(\text{"fantastic"}, \text{positive})}{\sum_{w \in V} count(w, \text{positive})} = 0 \tag{7.14}$$

But since naive Bayes naively multiplies all the feature likelihoods together, zero probabilities in the likelihood term for any class will cause the probability of the class to be zero, no matter the other evidence!

The simplest solution is the add-one (Laplace) smoothing introduced in Chapter 4. While Laplace smoothing is usually replaced by more sophisticated smoothing algorithms in language modeling, it is commonly used in naive Bayes text categorization:

$$\hat{P}(w_i|c) = \frac{count(w_i,c)+1}{\sum_{w\in V}(count(w,c)+1)} = \frac{count(w_i,c)+1}{\left(\sum_{w\in V} count(w,c)\right)+|V|} \tag{7.15}$$

What do we do about words that occur in our test data but are not in our vocabulary at all because they did not occur in any training document in any class? The standard solution for such **unknown words** is to ignore such words—remove them from the test document and not include any probability for them at all.

Fig. 7.2 shows the final algorithm.

function TRAIN NAIVE BAYES(D, C) **returns** log $P(c)$ and log $P(w|c)$

for each class $c \in C$ # Calculate $P(c)$ terms
  N_{doc} = number of documents in D
  N_c = number of documents from D in class c
  $logprior[c] \leftarrow \log \frac{N_c}{N_{doc}}$
  $V \leftarrow$ vocabulary of D
  $bigdoc[c] \leftarrow$ **append**(d) **for** d $\in$ D **with** class c
  for each word w in V # Calculate $P(w|c)$ terms
    $count(w,c) \leftarrow$ # of occurrences of w in $bigdoc[c]$
    $loglikelihood[w,c] \leftarrow \log \frac{count(w,c)+1}{\sum_{w' \text{ in } V} count(w',c)+1}$
return $logprior$, $loglikelihood$, V

function TEST NAIVE BAYES($testdoc$, $logprior$, $loglikelihood$, C, V) **returns** best c

for each class $c \in C$
  $sum[c] \leftarrow logprior[c]$
  for each position i in $testdoc$
    if $word[i] \in V$
      $sum[c] \leftarrow sum[c] + loglikelihood[word[i],c]$
return $\text{argmax}_c\ sum[c]$

Figure 7.2 The naive Bayes algorithm, using add-1 smoothing. To use add-α smoothing instead, change the $+1$ to $+\alpha$ for loglikelihood counts in training.

7.1.2 Worked example

Let's walk through an example of training and testing naive Bayes with add-one smoothing. We'll use a sentiment analysis domain with the two classes positive (+) and negative (-), and take the following miniature training and test documents simplified from actual movie reviews.

	Cat	Documents
Training	-	just plain boring
	-	entirely predictable and lacks energy
	-	no surprises and very few laughs
	+	very powerful
	+	the most fun film of the summer
Test	?	predictable with no originality

The prior $P(c)$ for the two classes is computed via Eq. 7.12 as $\frac{N_c}{N_{doc}}$:

$$P(-) = \frac{3}{5} \qquad P(+) = \frac{2}{5}$$

The likelihoods from the training set for the four words "predictable", "with", "no", and "originality", are as follows, from Eq. 7.15 (computing the probabilities for the remainder of the words in the training set is left as Exercise 7.**??**).

$$\begin{aligned}
P(\text{“predictable”}|-) &= \frac{1+1}{14+20} \qquad P(\text{“predictable”}|+) = \frac{0+1}{9+20} \\
P(\text{“with”}|-) &= \frac{0+1}{14+20} \qquad P(\text{“with”}|+) = \frac{0+1}{9+20} \\
P(\text{“no”}|-) &= \frac{1+1}{14+20} \qquad P(\text{“no”}|+) = \frac{0+1}{9+20} \\
P(\text{“originality”}|-) &= \frac{0+1}{14+20} \qquad P(\text{“originality”}|+) = \frac{0+1}{9+20}
\end{aligned}$$

For the test sentence S = "predictable with no originality", the chosen class, via Eq. 7.9, is therefore computed as follows:

$$\begin{aligned}
P(S|-)P(-) &= \frac{3}{5} \times \frac{2 \times 1 \times 2 \times 1}{34^4} = 1.8 \times 10^{-6} \\
P(S|+)P(+) &= \frac{2}{5} \times \frac{1 \times 1 \times 1 \times 1}{29^4} = 5.7 \times 10^{-7}
\end{aligned}$$

The model thus predicts the class *negative* for the test sentence.

7.1.3 Optimizing for Sentiment Analysis

While standard naive Bayes text classification can work well for sentiment analysis, some small changes are generally employed that improve performance.

First, for sentiment classification and a number of other text classification tasks, whether a word occurs or not seems to matter more than its frequency. Thus it often improves performance to clip the word counts in each document at 1. This variant is called **binary multinominal naive Bayes** or **binary NB**. The variant uses the same Eq. 7.10 except that for each document we remove all duplicate words before concatenating them into the single big document. Fig. 7.3 shows an example in which a set of four documents (shortened and text-normalized for this example) are remapped to binary, with the modified counts shown in the table on the right. The example is worked without add-1 smoothing to make the differences clearer. Note

binary NB

Four original documents:

- − it was pathetic the worst part was the boxing scenes
- − no plot twists or great scenes
- + and satire and great plot twists
- + great scenes great film

After per-document binarization:

- − it was pathetic the worst part boxing scenes
- − no plot twists or great scenes
- + and satire great plot twists
- + great scenes film

	NB Counts		Binary Counts	
	+	−	+	−
and	2	0	1	0
boxing	0	1	0	1
film	1	0	1	0
great	3	1	2	1
it	0	1	0	1
no	0	1	0	1
or	0	1	0	1
part	0	1	0	1
pathetic	0	1	0	1
plot	1	1	1	1
satire	1	0	1	0
scenes	1	2	1	2
the	0	2	0	1
twists	1	1	1	1
was	0	2	0	1
worst	0	1	0	1

Figure 7.3 An example of binarization for the binary naive Bayes algorithm.

that the results counts need not be 1; the word *great* has a count of 2 even for Binary NB, because it appears in multiple documents.

A second important addition commonly made when doing text classification for sentiment is to deal with negation. Consider the difference between *I really like this movie* (positive) and *I didn't like this movie* (negative). The negation expressed by *didn't* completely alters the inferences we draw from the predicate *like*. Similarly, negation can modify a negative word to produce a positive review (*don't dismiss this film*, *doesn't let us get bored*.

A very simple baseline that is commonly used in sentiment to deal with negation is during text normalization is to prepend the prefix *NOT_* to every word after a token of logical negation (*n't, not, no, never* until the next punctuation mark. Thus the phrase

```
didnt like this movie , but I
```

becomes

```
didnt NOT_like NOT_this NOT_movie , but I
```

Newly formed 'words' like *NOT_like*, *NOT_recommend* will thus occur more often in negative document and act as cues for negative sentiment, while words like *NOT_bored*, *NOT_dismiss* will acquire positive associations. We will return in Chapter 15 to the use of parsing to deal more accurately with the scope relationship between these negation words and the predicates they modify, but this simple baseline works quite well in practice.

Finally, in some situations we might have insufficient labeled training data to train accurate naive Bayes classifiers using all words in the training set to estimate positive and negative sentiment. In such cases we can instead derive the positive and negative word features from **sentiment lexicons**, lists of words that are pre-annotated with positive or negative sentiment. Four popular lexicons are the **General Inquirer** (Stone et al., 1966), **LIWC** (Pennebaker et al., 2007), the opinion lexicon of Hu and Liu (2004) and the MPQA Subjectivity Lexicon (Wilson et al., 2005).

sentiment lexicons

General Inquirer

LIWC

For example the MPQA subjectivity lexicon has 6885 words, 2718 positive and 4912 negative, each marked for whether are strongly or weakly biased. Some samples of positive and negative words from the MPQA lexicon include:

+ : *admirable, beautiful, confident, dazzling, ecstatic, favor, glee, great*
− : *awful, bad, bias, catastrophe, cheat, deny, envious, foul, harsh, hate*

?? will discuss how these lexicons can be learned automatically.

A common way to use lexicons in the classifier is to use as one feature the totaled number of occurrences of any words in the positive lexicon, and as a second feature the totaled number of occurrences of words in the negative lexicon. Using just two features results in classifiers that are much less sparse to small amounts of training data, and may generalize better.

7.1.4 Naive Bayes as a Language Model

Naive Bayes classifiers can use any sort of feature: dictionaries, URLs, email addresses, network features, phrases, parse trees, and so on. But if, as in the previous section, we use only individual word features, and we use all of the words in the text (not a subset) then naive Bayes has an important similarity to language modeling. Specifically, a naive Bayes model can be viewed as a set of class-specific unigram language models, in which the model for each class instantiates a unigram language model.

Since the likelihood features from the naive Bayes model assign a probability to each word $P(word|c)$, the model also assigns a probability to each sentence:

$$P(s|c) = \prod_{i \in positions} P(w_i|c) \tag{7.16}$$

Thus consider a naive Bayes model with the classes *positive* (+) and *negative* (-) and the following model parameters:

w	P(w|+)	P(w|-)
I	0.1	0.2
love	0.1	0.001
this	0.01	0.01
fun	0.05	0.005
film	0.1	0.1
...	...	...

Each of the two columns above instantiates a language model that can assign a probability to the sentence "I love this fun film":

$$\begin{aligned} P(\text{"I love this fun film"}|+) &= 0.1 \times 0.1 \times 0.01 \times 0.05 \times 0.1 = 0.0000005 \\ P(\text{"I love this fun film"}|-) &= 0.2 \times 0.001 \times 0.01 \times 0.005 \times 0.1 = .0000000010 \end{aligned}$$

As it happens, the positive model assigns a higher probability to the sentence: $P(s|pos) > P(s|neg)$. Note that this is just the likelihood part of the naive Bayes model; once we multiply in the prior a full naive Bayes model might well make a different classification decision.

7.2 Evaluation: Precision, Recall, F-measure

To introduce the methods for evaluating text classification, let's first consider a simple binary *detection* task: spam-detection. In this task our goal is to label every text

as being in the spam category ("positive") or not. For each item (document) we therefore need to know whether our system called it spam or not. We also need to know whether 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

To build a metric, consider the contingency table shown in Fig. 7.4. Each cell labels a set of possible outcomes. In the spam detection case, for example, true positives are the documents that are indeed spam (indicated by our human-created gold labels) and our system said they were spam.

To the bottom right of the table is the equation for *accuracy*. Although accuracy might seem a natural metric, we generally don't use it, because when the classes are unbalanced (as indeed they are with spam, which is the majority of email) we can get a high accuracy by doing nothing and just always returning 'positive'. But that's not very helpful if our eventual goal is find useful email. Similarly, if we're a company doing sentiment analysis with the goal of finding and addressing consumer complaints about our products, and even assuming we are a fantastic company with 99% positive comments, we don't want to ignore the 1% of cases where customers have complaints. Thus we need a metric that rewards us for finding correct examples of both classes even in unbalanced situations.

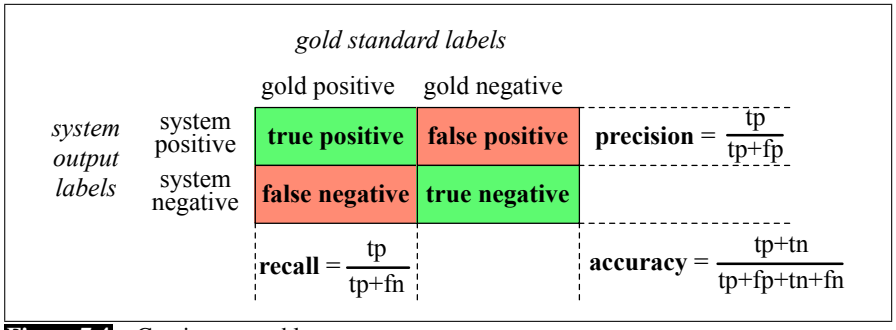

Figure 7.4 Contingency table

Instead, we most commonly report a combination of two metrics, **precision** and **recall**, each of which measures a different aspect of a useful solution.

precision

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

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

F-measure

The ***F*-measure** (van Rijsbergen, 1975) combines these two measures into a single metric, and is defined as

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

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

F1

$$F_1 = \frac{2PR}{P+R} \tag{7.17}$$

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

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, ..., a_n) = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \frac{1}{a_3} + ... + \frac{1}{a_n}} \tag{7.18}$$

and hence *F*-measure is

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1-\alpha}{\alpha}\right) \quad F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{7.19}$$

Harmonic mean is used because it is a conservative metric; 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.

7.2.1 More than two classes

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

any-of

There are two kinds of multi-class classification tasks. In **any-of** or **multi-label classification**, each document or item can be assigned more than one label. We can solve *any-of* classification by building separate binary classifiers for each class c, trained on positive examples labeled c and negative examples not labeled c. Given a test document or item d, then each classifier makes their decision independently, and we may assign multiple labels to d.

one-of
multinomial classification

More common in language processing is **one-of** or **multinomial classification**, in which the classes are mutually exclusive and each document or item appears in exactly one class. Here we again build a separate binary classifier trained on positive examples from c and negative examples from all other classes. Now given a test document or item d, we run all the classifiers and chose the label from the classifier with the highest score. Consider the sample confusion matrix for a hypothetical 3-way *one-of* email categorization decision (urgent, normal, spam) shown in Fig. 7.5.

The matrix shows, for example, that the system mistakenly labeled 1 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 contingency table, and then compute precision and recall from that table. Fig. 7.6 shows the contingency table for each class separately, and shows the computation of microaveraged and macroaveraged precision.

macroaveraging

microaveraging

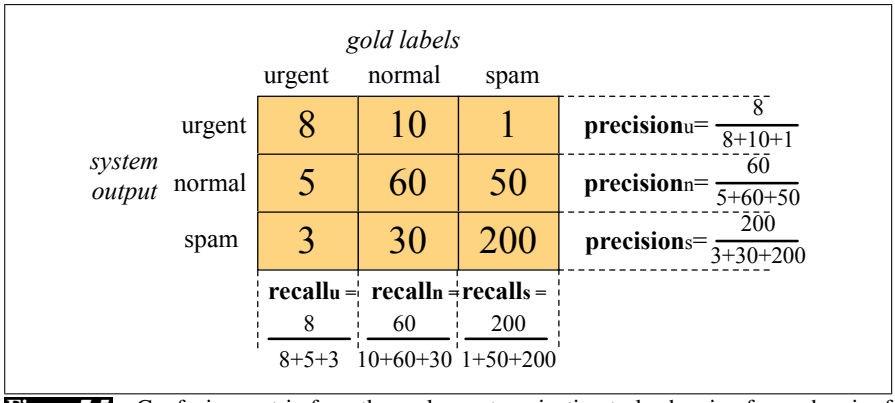

Figure 7.5 Confusion matrix for a three-class categorization task, showing for each pair of classes (c_1, c_2), how many documents from c_1 were (in)correctly assigned to c_2

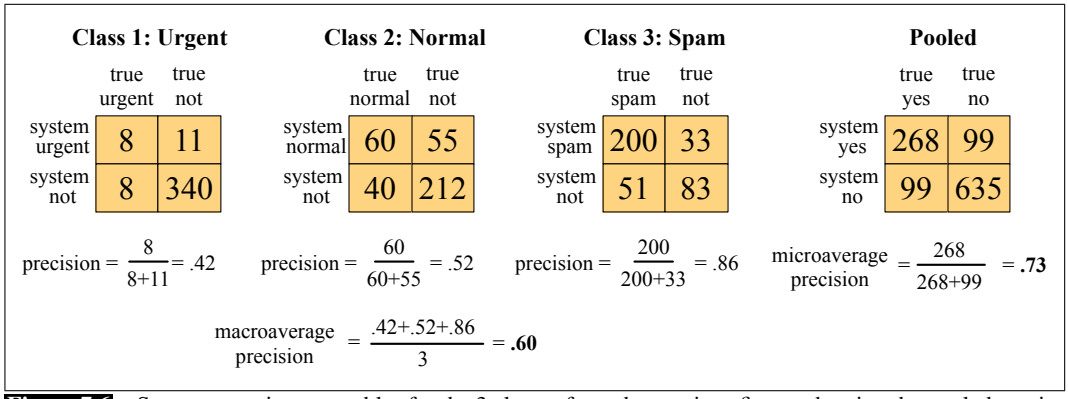

Figure 7.6 Separate contingency tables for the 3 classes from the previous figure, showing the pooled contingency table and the microaveraged and macroaveraged precision.

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.

7.2.2 Test sets and Cross-validation

Just as with language modeling (Section **??**) in text classification (and other applications of statistical classification in language processing) we use the training set to train our models. Then we use the **development test set** (also called a **dev-test** set) to perhaps tune some parameters, and in general decide what the best model is. Once we come up with what we think is the best model, we run it on the (hitherto unseen) test set to report its performance.

development test set
dev-test

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. It would be better if we could somehow use **all** our data both for training and test. We do this by **cross-validation**: we randomly choose a training and test set division of our data, train our classifier, and then compute the error rate on the test set. Then we repeat with a different randomly selected training set and test set. We do this sampling process 10 times and average these 10 runs to get an average error rate.

cross-validation

10-fold cross-validation

This is called **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. But looking at the corpus is often important for designing the system. 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. 7.7.

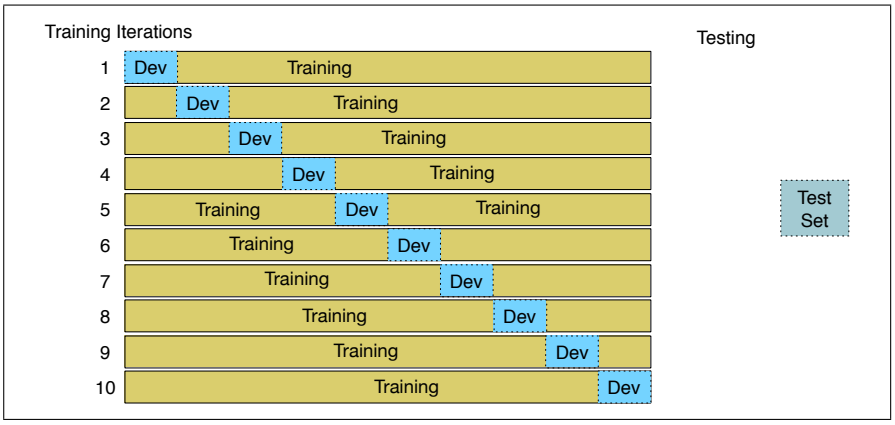

Figure 7.7 10-fold crossvalidation

7.3 Statistical Significance Testing

In building systems we are constantly comparing the performance of systems. Often we have added some new bells and whistles to our algorithm and want to compare the new version of the system to the unaugmented version. Or we want to compare our algorithm to a previously published one to know which is better.

We might image that to compare the performance of two classifiers A and B all we have to do is look at A and B's score on the same test set—for example for we might chose to compare macro-averaged F1— and see whether it's A or B that has the higher score. But just looking at this one difference isn't good enough, because A might have a better performance than B on a particular test set just by chance.

Let's say we have a test set x of n observations $x = x_1, x_2, .., x_n$ on which A's performance is better than B by $\delta(x)$. How can we know if A is really better than B?

null hypothesis

To do so we'd need to reject the **null hypothesis** that A isn't really better than B and this difference $\delta(x)$ occurred purely by chance. If the null hypothesis was correct, we would expect that if we had many test sets of size n and we measured A and B's performance on all of them, that on average A might accidentally still be better than B by this amount $\delta(x)$ just by chance.

More formally, if we had a random variable X ranging over test sets, the null hypothesis expects $P(\delta(X) > \delta(x)|H_0)$, the probability that we'll see similarly big differences just by chance, to be high.

If we had all these test sets we could just measure all the $\delta(x')$ for all the x'. If we found that those deltas didn't seem to be bigger than $\delta(x)$, that is, that p-value(x) was

sufficiently small, less than the standard thresholds of 0.05 or 0.01, then we might reject the null hypothesis and agree that $\delta(x)$ was a sufficiently surprising difference and A is really a better algorithm than B. Following (Berg-Kirkpatrick et al., 2012) we'll refer to $P(\delta(X) > \delta(x)|H_0)$ as p-value(x).

In language processing we don't generally use traditional statistical approaches like paired t-tests to compare system outputs because most metrics are not normally distributed, violating the assumptions of the tests. The standard approach to computing p-value(x) in natural language processing is to use non-parametric tests like the **bootstrap test** (Efron and Tibshirani, 1993)— which we will describe below—or a similar test, **approximate randomization** (Noreen, 1989). The advantage of these tests is that they can apply to any metric; from precision, recall, or F1 to the BLEU metric used in machine translation.

bootstrap test
approximate randomization

The intuition of the bootstrap is that we can actually create many pseudo test sets from one sample test set by treating the sample as the population and doing Monte-Carlo resampling from the sample. The method only makes the assumption that the 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. 7.8 shows the results on this test set, with each document 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 (Ⓑ) means that that classifier got the answer wrong. On the first document both A and B get the correct class (AB), while on the second document A got it right but B got it wrong (AⒷ). 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. To create each pseudo test set of size $N = 10$, we repeatedly (10 times) select a cell from row x with replacement. Fig. 7.8 shows a few examples.

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

Figure 7.8 The bootstrap: Examples of b pseudo test sets 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.

Now that we have a sampling distribution, we can do statistics. We'd like to know how often A beats B by more than $\delta(x)$ on each $x^{*(i)}$. But since the $x^{*(i)}$ were drawn from x, the expected value of $\delta(x^{*(i)}$ will lie very close to $\delta(x)$. To find out if A beats B by more than $\delta(x)$ on each pseudo test set, we'll need to shift the means of these samples by $\delta(x)$. Thus we'll be comparing for each $x^(i)$ whether $\delta(x^(i)) > 2\delta(x)$. The full algorithm for the bootstrap is shown in Fig. 7.9.

7.4 Multinomial Logistic Regression (MaxEnt Models)

We turn now to a second algorithm for classification called **multinomial logistic regression**, sometimes referred to within language processing as **maximum entropy**

function BOOTSTRAP(x, b) **returns** p-*value*(x)

Calculate $\delta(x)$
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 $\delta(x^{*(i)})$
for each $x^{*(i)}$
  $s \leftarrow s + 1$ **if** $\delta(x^{*(i)}) > 2\delta(x)$
p-value(x) $\approx \frac{s}{b}$
return p-value(x)

Figure 7.9 The bootstrap algorithm

MaxEnt
log-linear classifier

modeling, **MaxEnt** for short. Logistic regression belongs to the family of classifiers known as the **exponential** or **log-linear** classifiers. Like naive Bayes, it work by extracting some set of weighted features from the input, taking logs, and combining them linearly (meaning that each feature is multiplied by a weight and then added up). Technically, **logistic regression** refers to a classifier that classifies an observation into one of two classes, and **multinomial logistic** regression is used when classifying into more than two classes, although informally and in this chapter we sometimes use the shorthand **logistic regression** even when we are talking about multiple classes.

The most important difference between naive Bayes and logistic regression is that logistic regression is a **discriminative** classifier while naive Bayes is a **generative** classifier. To see what this means, recall that the job of a probabilistic classifier is to choose which output label y to assign an input x, choosing the y that maximizes $P(y|x)$. In the naive Bayes classifier, we used Bayes rule to estimate this best y indirectly from the likelihood $P(x|y)$ (and the prior $P(y)$):

$$\hat{y} = \operatorname*{argmax}_y P(y|x) = \operatorname*{argmax}_y P(x|y)P(y) \tag{7.20}$$

generative model

Because of this indirection, naive Bayes is a **generative model**: a model that is trained to **generate** the data x from the class y. The likelihood term $P(x|y)$ expresses that we are given the class y and are trying to predict which features we expect to see in the input x. Then we use Bayes rule to compute the probability we really want: $P(y|x)$.

discriminative model

But why not instead just directly compute $P(y|x)$? A **discriminative model** takes this direct approach, computing $P(y|x)$ by discriminating among the different possible values of the class y rather than first computing a likelihood:

$$\hat{y} = \operatorname*{argmax}_y P(y|x) \tag{7.21}$$

While logistic regression thus differs in the way it estimates probabilities, it is still like naive Bayes in being a linear classifier. Logistic regression estimates $P(y|x)$ by extracting some set of features from the input, combining them linearly (multiplying each feature by a weight and adding them up), and then applying a function to this combination.

We can't, however, just compute $P(y|x)$ directly from features and weights as follows:

$$P(y = C|x) = \sum_{i=1}^{N} w_i f_i \tag{7.22}$$

$$= w \cdot f \tag{7.23}$$

That's because the expression $\sum_{i=1}^{N} w_i \times f_i$ produces values from $-\infty$ to ∞; nothing in the equation above forces the output to be a legal probability, that is, to lie between 0 and 1. In fact, since weights are real-valued, might even be negative!

We'll solve this in two ways. First, we'll wrap the exp function around the weight-feature dot-product $w \cdot f$, which will make the values positive, and we'll create the proper denominator to make everything a legal probability and sum to 1.

$$p(c|x) = \frac{1}{Z} \exp \sum_{i} w_i f_i \tag{7.24}$$

Fleshing out this normalization factor Z , and specifying the number of features as N:

$$p(c|x) = \frac{\exp\left(\sum_{i=1}^{N} w_i f_i\right)}{\sum_{C} \exp\left(\sum_{i=1}^{N} w_i f_i\right)} \tag{7.25}$$

We need to make one more change to see the final logistic regression/MaxEnt equation. So far we've been assuming that the features f_i are real-valued, but it is more common in language processing to use binary-valued features. A feature that takes on only the values 0 and 1 is called an **indicator function**. Furthermore, the features are not just a property of the observation x, but are instead a property of both the observation x and the candidate output class c. Thus, in MaxEnt, instead of the notation f_i or $f_i(x)$, we use the notation $f_i(c,x)$, meaning feature i for a particular class c for a given observation x. The final equation for computing the probability of y being of class c given x in MaxEnt is then

indicator function

$$p(c|x) = \frac{\exp\left(\sum_{i=1}^{N} w_i f_i(c,x)\right)}{\sum_{c' \in C} \exp\left(\sum_{i=1}^{N} w_i f_i(c',x)\right)} \tag{7.26}$$

7.4.1 Features in Multinomial Logistic Regression

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

Suppose we are doing text classification, and we would like to know whether to assign the sentiment class $+$, $-$, or 0 (neutral) to a document. Here are five potential features, representing that the document x contains the word *great* and the class is

+ (f_1), contains the word *second-rate* and the class is + (f_2), and contains the word *no* and the class is − (f_3).

$$f_1(c,x) = \begin{cases} 1 & \text{if "great"} \in x \ \& \ c = + \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(c,x) = \begin{cases} 1 & \text{if "second-rate"} \in x \ \& \ c = - \\ 0 & \text{otherwise} \end{cases}$$

$$f_3(c,x) = \begin{cases} 1 & \text{if "no"} \in x \ \& \ c = - \\ 0 & \text{otherwise} \end{cases}$$

$$f_4(c,x) = \begin{cases} 1 & \text{if "enjoy"} \in x \ \& \ c = - \\ 0 & \text{otherwise} \end{cases}$$

Each of these features has a corresponding weight, which can be positive or negative. Weight $w_1(x)$ indicates the strength of *great* as a cue for class +, $w_2(x)$ and $w_3(x)$ the strength of *second-rate* and *no* for the class −. These weights would likely be positive—logically negative words like *no* or *nothing* turn out to be more likely to occur in documents with negative sentiment (Potts, 2011). Weight $w_4(x)$, the strength of *enjoy* for −, would likely have a negative weight. We'll discuss in the following section how these weights are learned.

Since each feature is dependent on both a property of the observation and the class being labeled, we would have additional features for the links between *great* and the negative class −, or *no* and the neutral class 0, and so on.

Similar features could be designed for other language processing classification tasks. For period disambiguation (deciding if a period is the end of a sentence or part of a word), we might have the two classes EOS (end-of-sentence) and not-EOS and features like f_1 below expressing that the current word is lower case and the class is EOS (perhaps with a positive weight), or that the current word is in our abbreviations dictionary ("Prof.") and the class is EOS (perhaps with a negative weight). A feature can also express a quite complex combination of properties. For example a period following a upper cased word is a likely to be an EOS, but if the word itself is *St.* and the previous word is capitalized, then the period is likely part of a shortening of the word *street*.

$$f_1(c,x) = \begin{cases} 1 & \text{if "}Case(w_i) = \text{Lower"} \ \& \ c = \text{EOS} \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(c,x) = \begin{cases} 1 & \text{if "}w_i \in \text{AcronymDict"} \ \& \ c = \text{EOS} \\ 0 & \text{otherwise} \end{cases}$$

$$f_3(c,x) = \begin{cases} 1 & \text{if "}w_i = \text{St."} \ \& \ \text{"}Case(w_{i-1}) = \text{Upper"} \ \& \ c = \text{EOS} \\ 0 & \text{otherwise} \end{cases}$$

In Chapter 8 we'll see features for the task of part-of-speech tagging. It's even possible to do discriminative language modeling as a classification task. In this case the set C of classes is the vocabulary of the language, and the task is to predict the next word using features of the previous words (traditional N-gram contexts). In that case, the features might look like the following, with a unigram feature for the word *the* (f_1) or *breakfast* (f_2), or a bigram feature for the context word *American* predicting *breakfast* (f_3). We can even create features that are very difficult to create in a traditional generative language model like predicting the word *breakfast* if the previous word ends in the letters *-an* like *Italian*, *American*, or *Malaysian* (f_4).

$$f_1(c,x) = \begin{cases} 1 & \text{if } \text{“}c = \text{the”} \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(c,x) = \begin{cases} 1 & \text{if } \text{“}c = \text{breakfast”} \\ 0 & \text{otherwise} \end{cases}$$

$$f_3(c,x) = \begin{cases} 1 & \text{if } \text{“}w_{i-1} = \text{American; \& } \ c = \text{breakfast”} \\ 0 & \text{otherwise} \end{cases}$$

$$f_4(c,x) = \begin{cases} 1 & \text{if } \text{“}w_{i-1}\text{ends in -an; \& } \ c = \text{breakfast”} \\ 0 & \text{otherwise} \end{cases}$$

feature templates

The features for the task of discriminative language models make it clear that we'll often need large numbers of features. Often these are created automatically via **feature templates**, abstract specifications of features. For example a trigram template might create a feature for every predicted word and pair of previous words in the training data. Thus the feature space is sparse, since we only have to create a feature if that n-gram exists 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.

7.4.2 Classification in Multinomial Logistic Regression

In logistic regression we choose a class by using Eq. 7.26 to compute the probability for each class and then choose the class with the maximum probability.

Fig. 7.10 shows an excerpt from a sample movie review in which the four feature defined in Eq. 7.27 for the two-class sentiment classification task are all 1, with the weights set as $w_1 = 1.9$, $w_2 = .9$, $w_3 = .7$, $w_4 = -.8$.

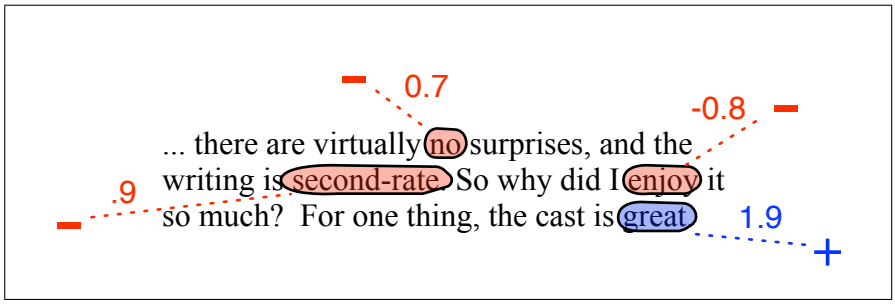

Figure 7.10 Some features and their weights for the positive and negative classes. Note the *negative* weight for *enjoy* meaning that it is evidence against the class negative $-$.

Given these 4 features and the input review x, $P(+|x)$ and $P(-|x)$ can be computed with Eq. 7.26:

$$P(+|x) = \frac{e^{1.9}}{e^{1.9} + e^{.9}e^{.7}e^{-.8}} = .82 \tag{7.27}$$

$$P(-|x) = \frac{e^{.9}e^{.7}e^{.8}}{e^{1.9} + e^{.9}e^{.7}e^{-.8}} = .18 \tag{7.28}$$

If the goal is just classification, we can even ignore the denominator and the exp and just choose the class with the highest dot product between the weights and features:

$$\begin{aligned}\hat{c} &= \operatorname*{argmax}_{c\in C} P(c|x) \\ &= \operatorname*{argmax}_{c\in C} \frac{\exp\left(\sum_{i=1}^{N} w_i f_i(c,x)\right)}{\sum_{c'\in C}\exp\left(\sum_{i=1}^{N} w_i f_i(c',x)\right)} \\ &= \operatorname*{argmax}_{c\in C} \exp\sum_{i=1}^{N} w_i f_i(c,x) \\ &= \operatorname*{argmax}_{c\in C} \sum_{i=1}^{N} w_i f_i(c,x) \end{aligned} \tag{7.29}$$

Computing the actual probability rather than just choosing the best class, however, is useful when the classifier is embedded in a larger system, as in a sequence classification domain like part-of-speech tagging (Section **??**).

Note that while the index in the inner sum of features in Eq. 7.29 ranges over the entire list of N features, in practice in classification it's not necessary to look at every feature, only the non-zero features. For text classification, for example, we don't have to consider features of words that don't occur in the test document.

7.4.3 Learning Logistic Regression

conditional maximum likelihood estimation

How are the parameters of the model, the weights w, learned? The intuition is to choose weights that make the classes of the training examples more likely. Indeed, logistic regression is trained with **conditional maximum likelihood estimation**. This means we choose the parameters w that maximize the (log) probability of the y labels in the training data given the observations x.

For an individual training observation $x^{(j)}$ in our training set (we'll use superscripts to refer to individual observations in the training set—this would be each individual document for text classification) the optimal weights are:

$$\hat{w} = \operatorname*{argmax}_{w} \log P(y^{(j)}|x^{(j)}) \tag{7.30}$$

For the entire set of observations in the training set, the optimal weights would then be:

$$\hat{w} = \operatorname*{argmax}_{w} \sum_{j} \log P(y^{(j)}|x^{(j)}) \tag{7.31}$$

The objective function L that we are maximizing is thus

$$L(w) = \sum_{j} \log P(y^{(j)}|x^{(j)})$$

$$= log \sum_j \frac{\exp\left(\sum_{i=1}^{N} w_i f_i(y^{(j)}, x^{(j)})\right)}{\sum_{y' \in Y} \exp\left(\sum_{i=1}^{N} w_i f_i(y'^{(j)}, x^{(j)})\right)}$$

$$= \log \sum_j \exp\left(\sum_{i=1}^{N} w_i f_i(y^{(j)}, x^{(j)})\right) - \log \sum_j \sum_{y' \in Y} \exp\left(\sum_{i=1}^{N} w_i f_i(y'^{(j)}, x^{(j)})\right)$$

Finding the weights that maximize this objective turns out to be a convex optimization problem, so we use hill-climbing methods like stochastic gradient ascent, L-BFGS (Nocedal 1980, Byrd et al. 1995), or conjugate gradient. Such gradient ascent methods start with a zero weight vector and move in the direction of the *gradient*, $L'(w)$, the partial derivative of the objective function $L(w)$ with respect to the weights. For a given feature dimension k, this derivative can be shown to be the difference between the following two counts:

$$L'(w) = \sum_j f_k(y^{(j)}, x^{(j)}) - \sum_j \sum_{y' \in Y} P(y'|x^{(j)}) f_k(y'^{(j)}, x^{(j)}) \tag{7.32}$$

These two counts turns out to have a very neat interpretation. The first is just the count of feature f_k in the data (the number of times f_k is equal to 1). The second is the expected count of f_k, under the probabilities assigned by the current model:

$$L'(w) = \sum_j \text{Observed count}(f_k) - \text{Expected count}(f_k) \tag{7.33}$$

Thus in optimal weights for the model the model's expected feature values match the actual counts in the data.

7.4.4 Regularization

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**.

overfitting

regularization

To avoid overfitting a **regularization** term is added to the objective function in Eq. 7.32. Instead of the optimization in Eq. 7.31, we optimize the following:

$$\hat{w} = \underset{w}{\text{argmax}} \sum_j \log P(y^{(j)}|x^{(j)}) - \alpha R(w) \tag{7.34}$$

where $R(w)$, the regularization term, is used to penalize large weights. Thus a setting of the weights that matches the training data perfectly, but uses lots of weights with high values to do so, will be penalized more than than a setting that matches the data a little less well, but does so using smaller weights.

L2 regularization

There are two common regularization terms $R(w)$. **L2 regularization** is a quadratic function of the weight values, named because is uses the (square of the) L2 norm of the weight values. The L2 norm, $||W||_2$, is the same as the **Euclidean distance**:

$$R(W) = ||W||_2^2 = \sum_{j=1}^{N} w_j^2 \tag{7.35}$$

The L2 regularized objective function becomes:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_{j} \log P(y^{(j)}|x^{(j)}) - \alpha \sum_{i=1}^{N} w_i^2 \tag{7.36}$$

L1 regularization

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

$$R(W) = ||W||_1 = \sum_{i=1}^{N} |w_i| \tag{7.37}$$

The L1 regularized objective function becomes:

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_{j} \log P(y^{(j)}|x^{(j)}) - \alpha \sum_{i=1}^{N} |w_i| \tag{7.38}$$

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

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

$$\frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(w_j - \mu_j)^2}{2\sigma_j^2}\right) \tag{7.39}$$

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

$$\hat{w} = \underset{w}{\operatorname{argmax}} \prod_{j}^{M} P(y^{(j)}|x^{(j)}) \times \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(w_i - \mu_j)^2}{2\sigma_j^2}\right) \tag{7.40}$$

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

$$\hat{w} = \underset{w}{\operatorname{argmax}} \sum_j \log P(y^{(j)}|x^{(j)}) - \alpha \sum_{i=1}^{N} w_i^2 \tag{7.41}$$

which is in the same form as Eq. 7.36.

7.5 Feature Selection

feature selection

The regularization technique introduced in the previous section is useful for avoiding overfitting by removing or downweighting features that are unlikely to generalize well. Many kinds of classifiers, however, including naive Bayes, do not have regularization, and so instead **feature selection** is used to choose the important features to keep and remove the rest. The basis of feature selection is to assign some metric of goodness to each feature, rank the features, and keep the best ones. The number of features to keep is a meta-parameter that can be optimized on a dev set.

mutual information

Features are generally ranked by how informative they are about the classification decision. A very common metric is **mutual information** (sometimes called **information gain**). It can be computed as follows (where $\bar{w}$ means a document doesn't contain the word w):

$$\begin{aligned} MI(w) = & -\sum_{i=1}^{C} P(c_i) \log P(c_i) \\ & + P(w) \sum_{i=1}^{C} P(c_i|w) \log P(c_i|w) \\ & + P(\bar{w}) \sum_{i=1}^{C} P(c_i|\bar{w}) \log P(c_i|\bar{w}) \end{aligned} \tag{7.42}$$

Other metrics for feature selection include χ^2, pointwise mutual information, and GINI index; see Yang and Pedersen (1997) for a comparison and Guyon and Elisseeff (2003) for a broad introduction survey of feature selection.

While feature selection is important for unregularized classifiers, it is sometimes also used in regularized classifiers in applications where speed is critical, since it is often possible to get equivalent performance with orders of magnitude fewer features.

7.6 Choosing a classifier and features

Logistic regression has a number of advantages over naive Bayes. The overly strong conditional independence assumptions of Naive Bayes mean that if two features are in fact correlated naive Bayes will multiply them both in as if they were independent, overestimating the evidence. Logistic regression is much more robust to correlated features; if two features f_1 and f_2 are perfectly correlated, regression will simply assign half the weight to w_1 and half to w_2.

Thus when there are many correlated features, logistic regression will assign a more accurate probability than naive Bayes. Nonetheless these less accurate probabilities often result nonetheless in naive Bayes making the correct classification decision. Furthermore, naive Bayes works extremely well (even better than logistic regression or SVMs) on small datasets (Ng and Jordan, 2002) or short documents (Wang and Manning, 2012). Furthermore, naive Bayes is easy to implement and very fast to train.

Nonetheless, algorithms like logistic regression and SVMs generally work better on larger documents or datasets.

bias-variance tradeoff
bias
variance

Classifier choice is also influenced by the **bias-variance tradeoff**. The **bias** of a classifier indicates how accurate it is at modeling different training sets. The **variance** of a classifier indicates how much its decisions are effected by small changes in training sets. Models with low bias (like SVMs with polynomial or RBF kernels) are very accurate at modeling the training data. Models with low variance (like naive Bayes) are likely to come to the same classification decision even from slightly different training data. But low-bias models tend to be so accurate at fitting the training data that they overfit, and do not generalize well to very different test sets. And low-variance models tend to generalize so well that they may not have sufficient accuracy. Thus any given model trades off bias and variance. Adding more features decreases bias by making it possible to more accurately model the training data, but increases variance because of overfitting. Regularization and feature selection are ways to improve (lower) the variance of classifier by downweighting or removing features that are likely to overfit.

In addition to the choice of a classifier, the key to successful classification is the design of appropriate features. Features are generally designed by examining the training set with an eye to linguistic intuitions and the linguistic literature on the domain. A careful error analysis on the training or dev set. of an early version of a system often provides insights into features.

For some tasks it is especially helpful to build complex features that are combinations of more primitive features. We saw such a features for period disambiguation above, where a period on the word *St.* was less likely to be the end of sentence if the previous word was capitalized. For logistic regression and naive Bayes these combination features or **feature interactions** have to be designed by hand.

feature interactions

Some other machine learning models can automatically model the interactions between features. For tasks where these combinations of features are important (especially when combination of categorical features and real-valued features might be helpful), the most useful classifiers may be such classifiers,including Support Vector Machines (**SVMs**) with polynomial or RBF kernels, and **random forests**. See the pointers at the end of the chapter.

SVMs
random forests

7.7 Summary

This chapter introduced the **naive Bayes** and multinomial **logistic regression** (**MaxEnt**) models for **classification** and applied them to the **text categorization** task of **sentiment analysis**.

- Many language processing tasks can be viewed as tasks of **classification**. learn to model the class given the observation.
- Text categorization, in which an entire text is assigned a class from a finite set,

comprises such tasks as **sentiment analysis**, **spam detection**, email classification, and authorship attribution.

- Sentiment analysis classifies a text as reflecting the positive or negative orientation (**sentiment**) that a writer expresses toward some object.
- Naive Bayes is a **generative** model that make the bag of words assumption (position doesn't matter) and the conditional independence assumption (words are conditionally independent of each other given the class)
- Naive Bayes with binarized features seems to work better for many text classification tasks.
- Multinomial logistic regression (also called MaxEnt or the Maximum Entropy classifier in language processing) is a discriminative model that assigns a class to an observation by computing a probability from an exponential function of a weighted set of features of the observation.
- **Regularization** is important in MaxEnt models for avoiding overfitting.
- **Feature selection** can be helpful in removing useless features to speed up training, and is also important in unregularized models for avoiding overfitting.

Bibliographical and Historical Notes

Multinomial naive Bayes text classification was proposed by Maron (1961) at the RAND Corporation for the task of assigning subject categories to journal abstracts. His model introduced most of the features of the modern form presented here, approximating the classification task with one-of categorization, and implementing add-δ smoothing and information-based feature selection.

The conditional independence assumptions of naive Bayes and the idea of Bayesian analysis of text seem to have been arisen multiple times. The same year as Maron's paper, Minsky (1961) proposed a naive Bayes classifier for vision and other artificial intelligence problems, and Bayesian techniques were also applied to the text classification task of authorship attribution by Mosteller and Wallace (1963). It had long been known that Alexander Hamilton, John Jay, and James Madison wrote the anonymously-published *Federalist* papers. in 1787–1788 to persuade New York to ratify the United States Constitution. Yet although some of the 85 essays were clearly attributable to one author or another, the authorship of 12 were in dispute between Hamilton and Madison. Mosteller and Wallace (1963) trained a Bayesian probabilistic model of the writing of Hamilton and another model on the writings of Madison, then computed the maximum-likelihood author for each of the disputed essays. Naive Bayes was first applied to spam detection in Heckerman et al. (1998).

Metsis et al. (2006), Pang et al. (2002), and Wang and Manning (2012) show that using boolean attributes with multinomial naive Bayes works better than full counts. Binary multinomial naive Bayes is sometimes confused with another variant of naive Bayes that also use a binary representation of whether a term occurs in a document: **Multivariate Bernoulli naive Bayes**. The Bernoulli variant instead estimates $P(w|c)$ as the fraction of documents that contain a term, and includes a probability for whether a term is *not* in a document McCallum and Nigam (1998) and Wang and Manning (2012) show that the multivariate Bernoulli variant of naive Bayes doesn't work as well as the multinomial algorithm for sentiment or other text tasks.

There are a variety of sources covering the many kinds of text classification tasks. There are a number of good overviews of sentiment analysis, including Pang and Lee (2008), and Liu and Zhang (2012). Stamatatos (2009) surveys authorship attribute algorithms, 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.

There are a number of good surveys of text classification (Manning et al. 2008, Aggarwal and Zhai 2012).

Maximum entropy modeling, including the use of regularization, was first applied to natural language processing (specifically machine translation) in the early 1990s at IBM (Berger et al. 1996, Della Pietra et al. 1997), and was soon applied to other NLP tasks like part-of-speech tagging and parsing (Ratnaparkhi 1996, Ratnaparkhi 1997) and text classification Nigam et al. (1999). See Chen and Rosenfeld (2000), Goodman (2004), and Dudík et al. (2007) on regularization for maximum entropy models.

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

Non-parametric methods for computing statistical significance were first introduced into natural language processing in the MUC computation (Chinchor et al., 1993). Our description of the bootstrap draws on the description in Berg-Kirkpatrick et al. (2012).

Exercises

7.1 Assume the following likelihoods for each word being part of a positive or negative movie review, and equal prior probabilities for each class.

	pos	neg
I	0.09	0.16
always	0.07	0.06
like	0.29	0.06
foreign	0.04	0.15
films	0.08	0.11

What class will Naive bayes assign to the sentence "I always like foreign films."?

7.2 Given the following short movie reviews, each labeled with a genre, either comedy or action:

1. fun, couple, love, love **comedy**
2. fast, furious, shoot **action**
3. couple, fly, fast, fun, fun **comedy**
4. furious, shoot, shoot, fun **action**
5. fly, fast, shoot, love **action**

and a new document D:

fast, couple, shoot, fly

compute the most likely class for D. Assume a naive Bayes classifier and use add-1 smoothing for the likelihoods.

7.3 Train two models, multinominal naive Bayes and binarized naive Bayes, both with add-1 smoothing, on the following document counts for key sentiment words, with positive or negative class assigned as noted.

doc	"good"	"poor"	"great"	(class)
d1.	3	0	3	pos
d2.	0	1	2	pos
d3.	1	3	0	neg
d4.	1	5	2	neg
d5.	0	2	0	neg

Use both naive Bayes models to assign a class (pos or neg) to this sentence:

A good, good plot and great characters, but poor acting.

Do the two models agree or disagree?

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

Bayes, T. (1763). *An Essay Toward Solving a Problem in the Doctrine of Chances*, Vol. 53. Reprinted in *Facsimiles of Two Papers by Bayes*, Hafner Publishing, 1963.

Berg-Kirkpatrick, T., Burkett, D., and Klein, D. (2012). An empirical investigation of statistical significance in NLP. In *EMNLP 2012*, pp. 995–1005.

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

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

Borges, J. L. (1964). *The analytical language of John Wilkins*. University of Texas Press. Trans. Ruth L. C. Simms.

Byrd, R. H., Lu, P., and Nocedal, J. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, *16*, 1190–1208.

Chen, S. F. and Rosenfeld, R. (2000). A survey of smoothing techniques for ME models. *IEEE Transactions on Speech and Audio Processing*, *8*(1), 37–50.

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

Della Pietra, S. A., Della Pietra, V. J., and Lafferty, J. D. (1997). Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *19*(4), 380–393.

Dudík, M., Phillips, S. J., and Schapire, R. E. (2007). Maximum entropy density estimation with generalized regularization and an application to species distribution modeling.. *Journal of Machine Learning Research*, *8*(6).

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

Goodman, J. (2004). Exponential priors for maximum entropy models. In *ACL-04*.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research*, *3*, 1157–1182.

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

Heckerman, D., Horvitz, E., Sahami, M., and Dumais, S. (1998). A bayesian approach to filtering junk e-mail. In *Proceeding of AAAI-98 Workshop on Learning for Text Categorization*, pp. 55–62.

Hu, M. and Liu, B. (2004). Mining and summarizing customer reviews. In *KDD*, pp. 168–177.

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

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

Maron, M. E. (1961). Automatic indexing: an experimental inquiry. *Journal of the ACM (JACM)*, *8*(3), 404–417.

McCallum, A. and Nigam, K. (1998). A comparison of event models for naive bayes text classification. In *AAAI/ICML-98 Workshop on Learning for Text Categorization*, pp. 41–48.

Metsis, V., Androutsopoulos, I., and Paliouras, G. (2006). Spam filtering with naive bayes-which naive bayes?. In *CEAS*, pp. 27–28.

Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the IRE*, *49*(1), 8–30.

Mosteller, F. and Wallace, D. L. (1963). Inference in an authorship problem: A comparative study of discrimination methods applied to the authorship of the disputed federalist papers. *Journal of the American Statistical Association*, *58*(302), 275–309.

Mosteller, F. and Wallace, D. L. (1964). *Inference and Disputed Authorship: The Federalist*. Springer-Verlag. A second edition appeared in 1984 as *Applied Bayesian and Classical Inference*.

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

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

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

Nocedal, J. (1980). Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, *35*, 773–782.

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

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

Pang, B., Lee, L., and Vaithyanathan, S. (2002). Thumbs up? Sentiment classification using machine learning techniques. In *EMNLP 2002*, pp. 79–86.

Pennebaker, J. W., Booth, R. J., and Francis, M. E. (2007). *Linguistic Inquiry and Word Count: LIWC 2007*. Austin, TX.

Potts, C. (2011). On the negativity of negation. In Li, N. and Lutz, D. (Eds.), *Proceedings of Semantics and Linguistic Theory 20*, pp. 636–659. CLC Publications, Ithaca, NY.

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

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

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

Stone, P., Dunphry, D., Smith, M., and Ogilvie, D. (1966). *The General Inquirer: A Computer Approach to Content Analysis*. Cambridge, MA: MIT Press.

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

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

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

Wilson, T., Wiebe, J., and Hoffmann, P. (2005). Recognizing contextual polarity in phrase-level sentiment analysis. In *HLT-EMNLP-05*, pp. 347–354.

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

Yang, Y. and Pedersen, J. O. (1997). A comparative study on feature selection in text categorization. In *ICML*, pp. 412–420.