

CHAPTER

4

Naive Bayes and Sentiment Classification

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; these are all examples of assigning a 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.

Many language processing tasks involve classification, although luckily our classes are much easier to define than those of Borges. In this chapter we introduce the naive Bayes algorithm and apply it to **text categorization**, the task of assigning a label or category to an entire text or document.

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. Extracting consumer or public sentiment is thus relevant for fields from marketing to politics.

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 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”.

language id

authorship attribution

Another thing we might want to know about a text is the language it's written in. Texts on social media, for example, can be in any number of languages and we'll need to apply different processing. The task of **language id** is thus the first step in most language processing pipelines. Related text classification tasks like **authorship attribution**—determining a text's author—are also relevant to the digital humanities, social sciences, and forensic linguistics.

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 essential for tasks below the level of the document as well. We've already seen period disambiguation (deciding if a period is the end of a sentence or part of a word), and 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. A part-of-speech tagger (Chapter 8) classifies each occurrence of a word in a sentence as, e.g., a noun or a verb.

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 handwritten rules. There are many areas of language processing where handwritten 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 instead done via **supervised machine learning**, and this will be the subject of the remainder of this chapter. In supervised learning, we have a data set of input observations, each associated with some correct output (a ‘supervision signal’). The goal of the algorithm is to learn how to map from a new observation to a correct output.

Formally, the task of supervised classification is to take an input x and a fixed set of output classes $Y = y_1, y_2, \dots, y_M$ 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 documents that have each been hand-labeled with a class: $(d_1, c_1), \dots, (d_N, c_N)$. 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. This chapter introduces naive Bayes; the following one introduces logistic regression. These exemplify two ways of doing classification. **Generative** classifiers like naive Bayes build a model of how a class could generate some input data. 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.

4.1 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.

The intuition of the classifier is shown in Fig. 4.1. 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 words *love*, *recommend*, and *movie* once, and so on.

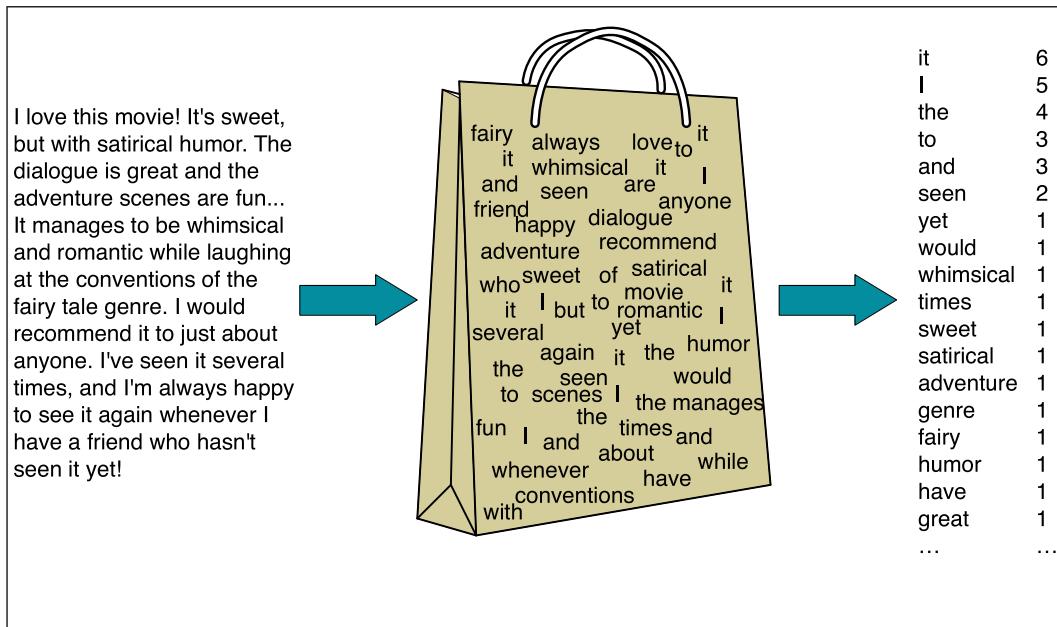


Figure 4.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

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. 4.1 we use the hat notation $\hat{\cdot}$ to mean “our estimate of the correct class”.

$$\hat{c} = \operatorname{argmax}_{c \in C} P(c|d) \quad (4.1)$$

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. 4.1 into other probabilities that have some useful properties. Bayes’ rule is presented in Eq. 4.2; it gives us a way to break down any conditional probability $P(x|y)$ into three other probabilities:

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)} \quad (4.2)$$

We can then substitute Eq. 4.2 into Eq. 4.1 to get Eq. 4.3:

$$\hat{c} = \operatorname{argmax}_{c \in C} P(c|d) = \operatorname{argmax}_{c \in C} \frac{P(d|c)P(c)}{P(d)} \quad (4.3)$$

We can conveniently simplify Eq. 4.3 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:

$$\hat{c} = \operatorname{argmax}_{c \in C} P(c|d) = \operatorname{argmax}_{c \in C} P(d|c)P(c) \quad (4.4)$$

We call Naive Bayes a **generative** model because we can read Eq. 4.4 as stating a kind of implicit assumption about how a document is generated: first a class is sampled from $P(c)$, and then the words are generated by sampling from $P(d|c)$. (In fact we could imagine generating artificial documents, or at least their word counts, by following this process). We'll say more about this intuition of generative models in Chapter 5.

To return to classification: we 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 $P(c)$ and the **likelihood** of the document $P(d|c)$:

$$\hat{c} = \operatorname{argmax}_{c \in C} \overbrace{P(d|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \quad (4.5)$$

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

$$\hat{c} = \operatorname{argmax}_{c \in C} \overbrace{P(f_1, f_2, \dots, f_n|c)}^{\text{likelihood}} \overbrace{P(c)}^{\text{prior}} \quad (4.6)$$

Unfortunately, Eq. 4.6 is still too hard to compute directly: without some simplifying assumptions, estimating the probability of every possible combination of features (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, \dots, f_n only encode word identity and not position.

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:

$$P(f_1, f_2, \dots, f_n|c) = P(f_1|c) \cdot P(f_2|c) \cdot \dots \cdot P(f_n|c) \quad (4.7)$$

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

$$c_{NB} = \operatorname{argmax}_{c \in C} P(c) \prod_{f \in F} P(f|c) \quad (4.8)$$

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:

positions \leftarrow all word positions in test document

$$c_{NB} = \operatorname{argmax}_{c \in C} P(c) \prod_{i \in \text{positions}} P(w_i|c) \quad (4.9)$$

**prior
probability
likelihood**

**naive Bayes
assumption**

Naive Bayes calculations, like calculations for language modeling, are done in log space, to avoid underflow and increase speed. Thus Eq. 4.9 is generally instead expressed as

$$c_{NB} = \operatorname{argmax}_{c \in C} \log P(c) + \sum_{i \in positions} \log P(w_i|c) \quad (4.10)$$

linear classifiers

By considering features in log space, Eq. 4.10 computes the predicted class as a linear function of input features. Classifiers that use a linear combination of the inputs to make a classification decision —like naive Bayes and also logistic regression—are called **linear classifiers**.

4.2 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 class 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:

$$\hat{P}(c) = \frac{N_c}{N_{doc}} \quad (4.11)$$

To 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:

$$\hat{P}(w_i|c) = \frac{\text{count}(w_i, c)}{\sum_{w \in V} \text{count}(w, c)} \quad (4.12)$$

Here the vocabulary V consists of the union of all the word types in all classes, not just the words in one class c .

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:

$$\hat{P}(\text{“fantastic”}|\text{positive}) = \frac{\text{count}(\text{“fantastic”}, \text{positive})}{\sum_{w \in V} \text{count}(w, \text{positive})} = 0 \quad (4.13)$$

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 3. While Laplace smoothing is usually replaced by more sophisticated smoothing

algorithms in language modeling, it is commonly used in naive Bayes text categorization:

$$\hat{P}(w_i|c) = \frac{\text{count}(w_i, c) + 1}{\sum_{w \in V} (\text{count}(w, c) + 1)} = \frac{\text{count}(w_i, c) + 1}{(\sum_{w \in V} \text{count}(w, c)) + |V|} \quad (4.14)$$

Note once again that it is crucial that the vocabulary V consists of the union of all the word types in all classes, not just the words in one class c (try to convince yourself why this must be true; see the exercise at the end of the chapter).

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 solution for such **unknown words** is to ignore them—remove them from the test document and not include any probability for them at all.

unknown word
stop words

Finally, some systems choose to completely ignore another class of words: **stop words**, very frequent words like *the* and *a*. This can be done by sorting the vocabulary by frequency in the training set, and defining the top 10–100 vocabulary entries as stop words, or alternatively by using one of the many predefined stop word lists available online. Then each instance of these stop words is simply removed from both training and test documents as if it had never occurred. In most text classification applications, however, using a stop word list doesn't improve performance, and so it is more common to make use of the entire vocabulary and not use a stop word list.

Fig. 4.2 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
     $bigdoc[c] \leftarrow \text{append}(d)$  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  $testdoc$ 
       $word \leftarrow testdoc[i]$ 
      if  $word \in V$ 
         $sum[c] \leftarrow sum[c] + loglikelihood[word, c]$ 
  return  $\text{argmax}_c sum[c]$ 

```

Figure 4.2 The naive Bayes algorithm, using add-1 smoothing. To use add- α smoothing instead, change the $+1$ to $+\alpha$ for loglikelihood counts in training.

4.3 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 fun

The prior $P(c)$ for the two classes is computed via Eq. 4.11 as $\frac{N_c}{N_{doc}}$:

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

The word *with* doesn't occur in the training set, so we drop it completely (as mentioned above, we don't use unknown word models for naive Bayes). The likelihoods from the training set for the remaining three words "predictable", "no", and "fun", are as follows, from Eq. 4.14 (computing the probabilities for the remainder of the words in the training set is left as an exercise for the reader):

$$\begin{aligned} P(\text{"predictable"}|-) &= \frac{1+1}{14+20} & P(\text{"predictable"}|+) &= \frac{0+1}{9+20} \\ P(\text{"no"}|-) &= \frac{1+1}{14+20} & P(\text{"no"}|+) &= \frac{0+1}{9+20} \\ P(\text{"fun"}|-) &= \frac{0+1}{14+20} & P(\text{"fun"}|+) &= \frac{1+1}{9+20} \end{aligned}$$

For the test sentence S = "predictable with no fun", after removing the word 'with', the chosen class, via Eq. 4.9, is therefore computed as follows:

$$\begin{aligned} P(-)P(S|-) &= \frac{3}{5} \times \frac{2 \times 2 \times 1}{34^3} = 6.1 \times 10^{-5} \\ P(+)P(S|+) &= \frac{2}{5} \times \frac{1 \times 1 \times 2}{29^3} = 3.2 \times 10^{-5} \end{aligned}$$

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

4.4 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 (see the end of the chapter for pointers to these results). This variant is called **binary**

binary NB or **multinomial naive Bayes** or **binary NB**. The variant uses the same Eq. 4.10 except that for each document we remove all duplicate words before concatenating them into the single big document. Fig. 4.3 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 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.

	NB Counts		Binary Counts	
	+	-	+	-
Four original documents:				
– it was pathetic the worst part was the boxing scenes	and	2	0	1 0
– no plot twists or great scenes	boxing	0	1	0 1
+ and satire and great plot twists	film	1	0	1 0
+ great scenes great film	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
– it was pathetic the worst part boxing scenes	plot	1	1	1 1
– no plot twists or great scenes	satire	1	0	1 0
+ and satire great plot twists	scenes	1	2	1 2
+ great scenes film	the	0	2	0 1
	twists	1	1	1 1
	was	0	2	0 1
	worst	0	1	0 1

Figure 4.3 An example of binarization for the binary naive Bayes algorithm.

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 analysis to deal with negation is the following: during text normalization, 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

didn't like this movie , but I

becomes

didn't 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 16 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

sentiment lexicons
General Inquirer
LIWC

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).

For example the MPQA subjectivity lexicon has 6885 words, 2718 positive and 4912 negative, each marked for whether it is 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

A common way to use lexicons in a naive Bayes classifier is to add a feature that is counted whenever a word from that lexicon occurs. Thus we might add a feature called ‘this word occurs in the positive lexicon’, and treat all instances of words in the lexicon as counts for that one feature, instead of counting each word separately. Similarly, we might add as a second feature ‘this word occurs in the negative lexicon’ of words in the negative lexicon. If we have lots of training data, and if the test data matches the training data, using just two features won’t work as well as using all the words. But when training data is sparse or not representative of the test set, using dense lexicon features instead of sparse individual-word features may generalize better.

We’ll return to this use of lexicons in Chapter 20, showing how these lexicons can be learned automatically, and how they can be applied to many other tasks beyond sentiment classification.

4.5 Naive Bayes for other text classification tasks

spam detection

In the previous section we pointed out that naive Bayes doesn’t require that our classifier use all the words in the training data as features. In fact features in naive Bayes can express any property of the input text we want.

Consider the task of **spam detection**, deciding if a particular piece of email is an example of spam (unsolicited bulk email)—one of the first applications of naive Bayes to text classification (Sahami et al., 1998).

A common solution here, rather than using all the words as individual features, is to predefine likely sets of words or phrases as features, combined with features that are not purely linguistic. For example the open-source SpamAssassin tool¹ defines features like the phrase “one hundred percent guaranteed”, or the feature *mentions millions of dollars*, which is a regular expression that matches suspiciously large sums of money. But it also includes features like *HTML has a low ratio of text to image area*, that aren’t purely linguistic and might require some sophisticated computation, or totally non-linguistic features about, say, the path that the email took to arrive. More sample SpamAssassin features:

- Email subject line is all capital letters
- Contains phrases of urgency like “urgent reply”
- Email subject line contains “online pharmaceutical”
- HTML has unbalanced “head” tags
- Claims you can be removed from the list

language id

For other tasks, like **language id**—determining what language a given piece

¹ <https://spamassassin.apache.org>

of text is written in—the most effective naive Bayes features are not words at all, but **character n-grams**, 2-grams ('zw') 3-grams ('nya', 'Vo'), or 4-grams ('ie z', 'thei'), or, even simpler **byte n-grams**, where instead of using the multibyte Unicode character representations called codepoints, we just pretend everything is a string of raw bytes. Because spaces count as a byte, byte n-grams can model statistics about the beginning or ending of words. A widely used naive Bayes system, `langid.py` ([Lui and Baldwin, 2012](#)) begins with all possible n-grams of lengths 1-4, using **feature selection** to winnow down to the most informative 7000 final features.

Language ID systems are trained on multilingual text, such as Wikipedia (Wikipedia text in 68 different languages was used in ([Lui and Baldwin, 2011](#))), or newswire. To make sure that this multilingual text correctly reflects different regions, dialects, and socioeconomic classes, systems also add Twitter text in many languages geo-tagged to many regions (important for getting world English dialects from countries with large Anglophone populations like Nigeria or India), Bible and Quran translations, slang websites like Urban Dictionary, corpora of African American Vernacular English ([Blodgett et al., 2016](#)), and so on ([Jurgens et al., 2017](#)).

4.6 Naive Bayes as a Language Model

As we saw in the previous section, naive Bayes classifiers can use any sort of feature: dictionaries, URLs, email addresses, network features, phrases, 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 $P(\text{word}|c)$, the model also assigns a probability to each sentence:

$$P(s|c) = \prod_{i \in \text{positions}} P(w_i|c) \quad (4.15)$$

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":

$$P(\text{"I love this fun film"}|+) = 0.1 \times 0.1 \times 0.01 \times 0.05 \times 0.1 = 0.0000005$$

$$P(\text{"I love this fun film"}|-) = 0.2 \times 0.001 \times 0.01 \times 0.005 \times 0.1 = .000000010$$

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.

4.7 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**.

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

In both cases, we need a metric for knowing how well our spam detector (or pie-tweet-detector) is doing. To evaluate any system for detecting things, we start by building a **confusion matrix** like the one shown in Fig. 4.4. 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		precision = $\frac{tp}{tp+fp}$
		gold positive	gold negative	
system output labels	system positive	true positive	false positive	precision = $\frac{tp}{tp+fp}$
	system negative	false negative	true negative	
		recall = $\frac{tp}{tp+fn}$		accuracy = $\frac{tp+tn}{tp+fp+tn+fn}$

Figure 4.4 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. 4.4: **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

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

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

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

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:

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

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 $F_{\beta=1}$ or just F_1 :

$$F_1 = \frac{2PR}{P + R} \quad (4.16)$$

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:

$$\text{HarmonicMean}(a_1, a_2, a_3, a_4, \dots, a_n) = \frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \frac{1}{a_3} + \dots + \frac{1}{a_n}} \quad (4.17)$$

and hence F-measure is

$$F = \frac{1}{\alpha \frac{1}{P} + (1 - \alpha) \frac{1}{R}} \quad \text{or} \left(\text{with } \beta^2 = \frac{1 - \alpha}{\alpha} \right) \quad F = \frac{(\beta^2 + 1)PR}{\beta^2P + R} \quad (4.18)$$

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.

4.7.1 Evaluating with more than two classes

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

		gold labels			
		urgent	normal	spam	
system output	urgent	8	10	1	$\text{precision}_u = \frac{8}{8+10+1}$
	normal	5	60	50	$\text{precision}_n = \frac{60}{5+60+50}$
	spam	3	30	200	$\text{precision}_s = \frac{200}{3+30+200}$
		$\text{recall}_u = \frac{8}{8+5+3}$	$\text{recall}_n = \frac{60}{10+60+30}$	$\text{recall}_s = \frac{200}{1+50+200}$	

Figure 4.5 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

macroaveraging
microaveraging

But we'll need to slightly modify our definitions of precision and recall. Consider the sample confusion matrix for a hypothetical 3-way *one-of* email categorization decision (urgent, normal, spam) shown in Fig. 4.5. 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. 4.6 shows the confusion matrix for each class separately, and shows the computation of 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.

4.8 Test sets and Cross-validation

development test set
devset

The training and testing procedure for text classification follows what we saw with language modeling (Section ??): we use the training set to train the model, then use the **development test set** (also called a **devset**) to perhaps tune some parameters,

Class 1: Urgent		Class 2: Normal		Class 3: Spam		Pooled	
true	true	true	true	true	true	true	true
urgent	not	normal	not	spam	not	yes	no
system urgent	8	11	system normal	60	55	system spam	200
system not	8	340	system not	40	212	system not	33
						51	83
						268	99
						99	635

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

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

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

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

$\text{macroaverage precision} = \frac{.42+.52+.86}{3} = .60$

Figure 4.6 Separate confusion matrices for the 3 classes from the previous figure, showing the pooled confusion matrix and the microaveraged and macroaveraged precision.

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

folds

10-fold 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**.

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. 4.7.

4.9 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