Chapter 2
From Textual Information to Numerical Vectors

To mine text, we first need to process it into a form that data-mining procedures can use. As mentioned in the previous chapter, this typically involves generating features in a spreadsheet format. Classical data mining looks at highly structured data. Our spreadsheet model is the embodiment of a representation that is supportive of predictive modeling. In some ways, predictive text mining is simpler and more restrictive than open-ended data mining. Because predictive mining methods are so highly developed, most time spent on data-mining projects is for data preparation. We say that text mining is unstructured because it is very far from the spreadsheet model that we need to process data for prediction. Yet, the transformation of data from text to the spreadsheet model can be highly methodical, and we have a carefully organized procedure to fill in the cells of the spreadsheet. First, of course, we have to determine the nature of the columns (i.e., the features) of the spreadsheet. Some useful features are easy to obtain (e.g., a word as it occurs in text) and some are much more difficult (e.g., the grammatical function of a word in a sentence such as subject, object, etc.). In this chapter, we will discuss how to obtain the kinds of features commonly generated from text.

2.1 Collecting Documents

Clearly, the first step in text mining is to collect the *data* (i.e., the relevant documents). In many text-mining scenarios, the relevant documents may already be given or they may be part of the problem description. For example, a Web page retrieval application for an intranet implicitly specifies the relevant documents to be the Web pages on the intranet. If the documents are readily identified, then they can be obtained, and the main issue is to cleanse the samples and ensure that they are of high quality. As with nontextual data, human intervention can compromise the integrity of the document collection process, and hence extreme care must be exercised. Sometimes, the documents may be obtained from document warehouses or databases. In these scenarios, it is reasonable to expect that data cleansing was done before deposit and we can be confident in the quality of the documents.

S.M. Weiss et al., *Fundamentals of Predictive Text Mining,*
Texts in Computer Science 41,
DOI 10.1007/978-1-84996-226-1_2,

In some applications, one may need to have a data collection process. For instance, for a Web application comprising a number of autonomous Web sites, one may deploy a software tool such as a Web crawler that collects the documents. In other applications, one may have a logging process attached to an input data stream for a length of time. For example, an e-mail audit application may log all incoming and outgoing messages at a mail server for a period of time.

Sometimes the set of documents can be extremely large and data-sampling techniques can be used to select a manageable set of relevant documents. These sampling techniques will depend on the application. For instance, documents may have a time stamp, and more recent documents may have a higher relevance. Depending on our resources, we may limit our sample to documents that are more useful.

For research and development of text-mining techniques, more generic data may be necessary. This is usually called a corpus. For the accompanying software, we mainly used the collection of Reuters news stories, referred to as Reuters corpus RCV1, obtainable from the Reuters Corporation Web site. However, there are many other corpora available that may be more appropriate for some studies.

In the early days of text processing (1950s and 1960s), one million words was considered a very large collection. This was the size of one of the first widely available collections, the Brown corpus, consisting of 500 samples of about 2000 words each of American English texts of varying genres. A European corpus, the Lancaster-Oslo-Bergen corpus (LOB), was modeled on the Brown corpus but was for British English. Both these are still available and still used. In the 1970s and 1980s, many more resources became available, some from academic initiatives and others as a result of government-sponsored research. Some widely used corpora are the Penn Tree Bank, a collection of manually parsed sentences from the Wall Street Journal; the TREC (Text Retrieval and Evaluation Conferences) collections, consisting of selections from the Wall Street Journal, the New York Times, Ziff-Davis Publications, the Federal Register, and others; the proceedings of the Canadian Parliament in parallel English–French translations, widely used in statistical machine translation research; and the Gutenberg Project, a very large collection of literary and other texts put into machine-readable form as the material comes out of copyright. A collection of Reuters news stories called Reuters-21578 Distribution 1.0 has been widely used in studying methods for text categorization.

As the importance of large text corpora became evident, a number of organizations and initiatives arose to coordinate activity and provide a distribution mechanism for corpora. Two of the main ones are the Linguistic Data Consortium (LDC) housed at the University of Pennsylvania and the International Computer Archive of Modern and Medieval English (ICAME), which resides in Bergen, Norway. Many other centers of varying size exist in academic institutions. The Text Encoding Initiative (TEI) is a standard for text collections sponsored by a number of professional societies concerned with language processing. There a number of Web sites devoted to corpus linguistics, most having links to collections, courses, software, etc.

Another resource to consider is the World Wide Web itself. Web crawlers can build collections of pages from a particular site, such as Yahoo, or on a particular topic. Given the size of the Web, collections built this way can be huge. The main

problem with this approach to document collection is that the data may be of dubious quality and require extensive cleansing before use. A more focused corpus can be built from the archives of USENET news groups and accessible from many ISPs directly or through Google Groups. These discussion groups cover a single topic, such as fly fishing, or broader topics such as the cultures of particular countries. A similar set of discussions is available from LISTSERVs. These are almost always available only by subscribing to a particular group but have the advantage that many lists have long-term archives.

Finally, institutions such as government agencies and corporations often have large document collections. Corporate collections are usually not available outside the corporation, but government collections often are. One widely studied collection is the MEDLINE data set from the National Institutes of Health, which contains a very large number of abstracts on medical subjects. The advantage of getting documents from such sources is that one can be reasonably sure that the data have been reviewed and are of good quality.

2.2 Document Standardization

Once the documents are collected, it is not uncommon to find them in a variety of different formats, depending on how the documents were generated. For example, some documents may have been generated by a word processor with its own proprietary format; others may have been generated using a simple text editor and saved as ASCII text; and some may have been scanned and stored as images. Clearly, if we are to process all the documents, it's helpful to convert them to a standard format.

The computer industry as a whole, including most of the text-processing community, has adopted XML (Extensible Markup Language) as its standard exchange format, and this is the standard we adopt for our document collections as well. Briefly, XML is a standard way to insert tags onto a text to identify its parts. Although tags can be nested within other tags to arbitrary depth, we will use that capability only sparingly here. We assume that each document is marked off from the other documents in the corpus by having a distinguishing tag at the beginning, such as <DOC>. By XML convention, tags come in beginning and ending pairs. They are enclosed in angle brackets, and the ending tag has a back slash immediately following the opening angle bracket. Within a document, there can be many other tags to mark off sections of the document. Common sections are <DATE>, <SUBJECT>, <TOPIC>, and <TEXT>. We will focus mainly on <SUBJECT>, <TOPIC>, and <TEXT>. The names are arbitrary. They could just as well be <HEADLINE> and <BODY>. An example of an XML document is shown in Fig. 2.1, where the document has a distinguishing tag of <DOC>.

Many currently available corpora are already in this format (e.g., the newer corpora available from Reuters). The main reason for identifying the pieces of a document consistently is to allow selection of those parts that will be used to generate features. We will almost always want to use the part delimited as <TEXT> but may also want to include parts marked <SUBJECT>, <HEADLINE>, or the like. Additionally, for text classification or clustering, one wants to generate features from a

```
<DOC>
<TEXT>
<TITLE>
Solving Regression Problems with Rule-based Classifiers
</TITLE>
<AUTHORS>
<AUTHOR>
Nitin Indurkhya
</AUTHOR>
<AUTHOR>
Sholom M. Weiss
</AUTHOR>
</AUTHORS>
<ABSTRACT>
We describe a lightweight learning method that induces an ensemble
of decision-rule solutions for regression problems. Instead of
direct prediction of a continuous output variable, the method
discretizes the variable by k-means clustering and solves the
resultant classification problem. Predictions on new examples are
made by averaging the mean values of classes with votes that are
close in number to the most likely class. We provide experimental
evidence that this indirect approach can often yield strong
results for many applications, generally outperforming direct
approaches such as regression trees and rivaling bagged regression
trees.
</ABSTRACT>
</TEXT>
</DOC>
```

Fig. 2.1 An XML document

TOPIC section if there is one. Selected document parts may be concatenated into a single string of characters or may be kept separate if one wants to distinguish the features generated from the headline, say, from those generated from the document body, and perhaps weight them differently.

Many word processors these days allow documents to be saved in XML format, and stand-alone filters can be obtained to convert existing documents without having to process each one manually. Documents encoded as images are harder to deal with currently. There are some OCR (optical character recognition) systems that can be useful, but these can introduce errors in the text and must be used with care.

Why should we care about document standardization? The main advantage of standardizing the data is that the mining tools can be applied without having to consider the pedigree of the document. For harvesting information from a document, it is irrelevant what editor was used to create it or what the original format was. The software tools need to read data just in one format, and not in the many different formats they came in originally.

2.3 Tokenization

Assume the document collection is in XML format and we are ready to examine the unstructured text to identify useful features. The first step in handling text is to break the stream of characters into words or, more precisely, *tokens*. This is fundamental

to further analysis. Without identifying the tokens, it is difficult to imagine extracting higher-level information from the document. Each token is an instance of a *type*, so the number of tokens is much higher than the number of types. As an example, in the previous sentence there are two tokens spelled "the." These are both instances of a type "the," which occurs twice in the sentence. Properly speaking, one should always refer to the frequency of occurrence of a type, but loose usage also talks about the frequency of a token. Breaking a stream of characters into tokens is trivial for a person familiar with the language structure. A computer program, though, being linguistically challenged, would find the task more complicated. The reason is that certain characters are sometimes token delimiters and sometimes not, depending on the application. The characters space, tab, and newline we assume are always delimiters and are not counted as tokens. They are often collectively called *white space*. The characters ( ) < > ! ? " are always delimiters and may also be tokens. The characters . , : - ' may or may not be delimiters, depending on their environment.

A period, comma, or colon between numbers would not normally be considered a delimiter but rather part of the number. Any other comma or colon is a delimiter and may be a token. A period can be part of an abbreviation (e.g., if it has a capital letter on both sides). It can also be part of an abbreviation when followed by a space (e.g., Dr.). However, some of these are really ends of sentences. The problem of detecting when a period is an end of sentence and when it is not will be discussed later. For the purposes of tokenization, it is probably best to treat any ambiguous period as a word delimiter and also as a token.

The apostrophe also has a number of uses. When preceded and followed by nondelimiters, it should be treated as part of the current token (e.g., isn't or D'angelo). When followed by an unambiguous terminator, it might be a closing internal quote or might indicate a possessive (e.g., Tess'). An apostrophe preceded by a terminator is unambiguously the beginning of an internal quote, so it is possible to distinguish the two cases by keeping track of opening and closing internal quotes.

A dash is a terminator and a token if preceded or followed by another dash. A dash between two numbers might be a subtraction symbol or a separator (e.g., 555-1212 as a telephone number). It is probably best to treat a dash not adjacent to another dash as a terminator and a token, but in some applications it might be better to treat the dash, except in the double dash case, as simply a character.

An example of pseudocode for tokenization is shown in Fig. 2.2. A version of this is available in the accompanying software.

To get the best possible features, one should always customize the tokenizer for the available text—otherwise extra work may be required after the tokens are obtained. The reader should note that the tokenization process is language-dependent. We, of course, focus on documents in English. For other languages, although the general principles will be the same, the details will differ.

2.4 Lemmatization

Once a character stream has been segmented into a sequence of tokens, the next possible step is to convert each of the tokens to a standard form, a process usually

```
Initialize:
   Set Stream to the input text string
   Set currentPosition to 0 and internalQuoteFlag to false
   Set delimiterSet to ',.;:!?()<>+"\n\t space
   Set whiteSpace to \t\n space
Procedure getNextToken:
   L1: cursor := currentPosition; ch := charAt(cursor);
      If ch = endOfStream then return null; endif
   L2: while ch is not endOfStream nor instanceOf(delimiterSet) do
         increment cursor by 1; ch := charAt(cursor);
      endwhile
      If ch = endOfStream then
         If cursor = currentPosition then return null; endif
      endif
      If ch is whiteSpace then
         If currentPosition = cursor then
            increment currentPosition by 1 and goto L1;
         else
            Token := substring(Stream,currentPosition,cursor-1);
            currentPosition := cursor+1; return Token;
         endif
      endif
      If ch = ' then
         If charAt(cursor-1) = instanceOf(delimiterSet) then
            internalQuoteFlag := true; increment currentPosition by 1; goto L1;
         endif
         If charAt(cursor+1) != instanceof(delimiterSet) then
            increment cursor by 1; ch := charAt(cursor); goto L2;
         elseif internalQuoteFlag = true then
            Token := substring(Stream,currentPosition,cursor-1);
            internalQuoteFlag := false;
         else
            Token := substring(Stream,currentPosition,cursor);
         endif
         currentPosition := cursor+1; return Token;
      endif
      If cursor = currentPosition then
         Token := ch; currentPosition := cursor+1;
      else
         Token := substring(Stream,currentPosition,cursor-1);
         currentPosition := cursor;
      endif
      return Token;
endprocedure
```

Fig. 2.2 Tokenization algorithm

referred to as *stemming* or *lemmatization*. Whether or not this step is necessary is application-dependent. For the purpose of document classification, stemming can provide a small positive benefit in some cases. Notice that one effect of stemming is to reduce the number of distinct types in a text corpus and to increase the frequency of occurrence of some individual types. For example, in the previous sentence, the

two instances of "types" would be reduced to the stem "type" and would be counted as instances of that type, along with instances of the tokens "type" and "typed." For classification algorithms that take frequency into account, this can sometimes make a difference. In other scenarios, the extra processing may not provide any significant gains.

2.4.1 Inflectional Stemming

In English, as in many other languages, words occur in text in more than one form. Any native English speaker will agree that the nouns "book" and "books" are two forms of the same word. Often, but not always, it is advantageous to eliminate this kind of variation before further processing (i.e., to normalize both words to the single form "book"). When the normalization is confined to regularizing grammatical variants such as singular/plural and present/past, the process is called "inflectional stemming." In linguistic terminology, this is called "morphological analysis." In some languages, for example Spanish, morphological analysis is comparatively simple. For a language such as English, with many irregular word forms and nonintuitive spelling, it is more difficult. There is no simple rule, for example, to bring together "seek" and "sought." Similarly, the stem for "rebelled" is "rebel," but the stem for "belled" is "bell." In other languages, inflections can take the form of infixing, as, in the German "angeben" (declare), for which the past participle is "angegeben."

Returning to English, an algorithm for inflectional stemming must be part rule-based and part dictionary-based. Any stemming algorithm for English that operates only on tokens, without more grammatical information such as part-of-speech, will make some mistakes because of ambiguity. For example, is "bored" the adjective as in "he is bored" or is it the past tense of the verb "bore"? Furthermore, is the verb "bore" an instance of the verb "bore a hole," or is it the past tense of the verb "bear"? In the absence of some often complicated disambiguation process, a stemming algorithm should probably pick the most frequent choice. Pseudocode for a somewhat simplified inflectional stemmer for English is given in Fig. 2.3. Notice how the algorithm consists of rules that are applied in sequence until one of them is satisfied. Also notice the frequent referrals to a dictionary, usually referred to as a *stemming dictionary*. Although the inflectional stemmer is not expected to be perfect, it will correctly identify quite a significant number of stems. An inflectional stemmer is available with the accompanying software.

2.4.2 Stemming to a Root

Some practitioners have felt that normalization more aggressive than inflectional stemming is advantageous for at least some text-processing applications. The intent of these stemmers is to reach a root form with no inflectional or derivational prefixes

Input: a text token and a dictionary
Doubling consonants: b d g k m n p r l t

Rules:
If token length < 4 **return** token
If token is number **return** token
If token is acronym **return** token
If token in dictionary **return** the stored stem
If token ends in s'
strip the ' and **return** stripped token
If token ends in 's
strip the 's and **return** stripped token
If token ends in "is", "us", or "ss" **return** token
If token ends in s
strip s, check in dictionary, and **return** stripped token if there
If token ends with es
strip es, check in dictionary, and **return** stripped token if there
If token ends in ies
replace ies by y and **return** changed token
If token ends in s
strip s and **return** stripped token
If token doesn't end with ed or ing **return** token
If token ends with ed
strip ed, check in dictionary and **return** stripped token if there
If token ends in ied
replace ied by y and **return** changed token
If token ends in eed
remove d and **return** stripped token if in dictionary
If token ends with ing
strip ing (if length > 5) and **return** stripped token if in dictionary
If token ends with ing and length ≤ 5 **return** token
// Now we have SS, the stripped stem, without ed or ing and it's
// not in the dictionary (otherwise algorithm would terminate)
If SS ends in doubling consonant
strip final consonant and **return** the changed SS if in dictionary
If doubling consonant was l **return** original SS
If no doubled consonants in SS
add e and **return** changed SS if in dictionary
If SS ends in c or z, or there is a g or l before the final doubling consonant
add e and **return** changed SS
If SS ends in any consonant that is preceded by a single vowel
add e and **return** changed SS
return SS

Fig. 2.3 Inflectional stemming algorithm

and suffixes. For example, "denormalization" is reduced to the stem "norm." The end result of such aggressive stemming is to reduce the number of types in a text collection very drastically, thereby making distributional statistics more reliable. Additionally, words with the same core meaning are coalesced, so that a concept such as "apply" has only one stem, although the text may have "reapplied", "applications", etc. We cannot make any broad recommendations as to when or when

not to use such stemmers. The usefulness of stemming is very much application-dependent. When in doubt, it doesn't hurt to try both with and without stemming if one has the resources to do so.

2.5 Vector Generation for Prediction

Consider the problem of categorizing documents. The characteristic features of documents are the tokens or words they contain. So without any deep analysis of the linguistic content of the documents, we can choose to describe each document by features that represent the most frequent tokens. Figure 2.4 describes this process. A version of this process is available in the accompanying software.

The collective set of features is typically called a *dictionary*. The tokens or words in the dictionary form the basis for creating a spreadsheet of numeric data corresponding to the document collection. Each row is a document, and each column represents a feature. Thus, a cell in the spreadsheet is a measurement of a feature (corresponding to the column) for a document (corresponding to a row). We will soon introduce the predictive methods that learn from such data. But let us first explore the various nuances of this data model and how it might influence the learning methods. In the most basic model of such data, we simply check for the presence or absence of words, and the cell entries are binary entries corresponding to a document and a word. The dictionary of words covers all the possibilities and corresponds to the number of columns in the spreadsheet. The cells will all have ones or zeros, depending on whether the words were encountered in the document.

If a learning method can deal with the high dimensions of such a global dictionary, this simple model of data can be very effective. Checking for words is simple

```
Input:
   ts, all the tokens in the document collection
   k, the number of features desired
Output:
   fs, a set of k features
Initialize:
   hs := empty hashtable

for each tok in ts do
   If hs contains tok then
      i := value of tok in hs
      increment i by 1
   else
      i := 1
   endif
   store i as value of tok in hs
endfor
sk := keys in hs sorted by decreasing value
fs := top k keys in sk
output fs
```

Fig. 2.4 Generating features from tokens

Table 2.1 Dictionary reduction techniques

Local dictionary
Stopwords
Frequent words
Feature selection
Token reduction: stemming, synonyms

because we do not actually check each word in the dictionary. We build a hash table of the dictionary words and see whether the document's words are in the hash table. Large samples of digital documents are readily available. This gives us confidence that many variations and combinations of words will show up in the sample. This expectation argues for spending less computational time preparing the data to look for similar words or remove weak words. Let the speedy computer find its own way during the learning process.

But, in many circumstances, we may want to work with a smaller dictionary. The sample may be relatively small, or a large dictionary may be unwieldy. In such cases, we might try to reduce the size of the dictionary by various transformations of a dictionary and its constituent words. Depending on the learning method, many of these transformations can improve predictive performance. Table 2.1 lists some of the transformations that can be performed.

If prediction is our goal, we need one more column for the correct answer (or class) for each document. In preparing data for a learning method, this information will be available from the document labels. Our labels are generally binary, and the smaller class is almost always the one of interest. Instead of generating a global dictionary for both classes, we may consider only words found in the class that we are trying to predict. If this class is far smaller than the negative class, which is typical, such a *local dictionary* will be far smaller than the global dictionary.

Another obvious reduction in dictionary size is to compile a list of *stopwords* and remove them from the dictionary. These are words that almost never have any predictive capability, such as articles *a* and *the* and pronouns such as *it* and *they*. These common words can be discarded before the feature generation process, but it's more effective to generate the features first, apply all the other transformations, and at the very last stage reject the ones that correspond to stopwords.

Frequency information on the word counts can be quite useful in reducing dictionary size and can sometimes improve predictive performance for some methods. The most frequent words are often stopwords and can be deleted. The remaining most frequently used words are often the important words that should remain in a local dictionary. The very rare words are often typos and can also be dismissed. For some learning methods, a local dictionary of the most frequent words, perhaps less than 200, can be surprisingly effective.

An alternative approach to local dictionary generation is to generate a global dictionary from all documents in the collection. Special feature selection routines will attempt to select a subset of words that appear to have the greatest potential for prediction. These selection methods are often complicated and independent of the prediction method. Generally, we do not use them and rely on just frequency

information, which is quite easy to determine. Any of the feature selection methods that have been used in alternative statistical or machine-learning settings may be tried. Many of these have been developed for real variables and without an emphasis on discrete or binary attributes. Some text-specific methods will be described later on in Sect. 2.5.3, but many of the prediction methods have already been adjusted for text to deal with larger dictionaries rather than repeatedly generating smaller dictionaries. If many classes must be determined, then the generation of a smaller dictionary must be repeated for each prediction problem. For example, if we have 100 topics to categorize, then we have 100 binary prediction problems to solve. Our choices are 100 small dictionaries or one big one. Typically, the vectors implied by a spreadsheet model will also be regenerated to correspond to the small dictionary.

Instead of placing every possible word in the dictionary, we might follow the path of the printed dictionary and avoid storing every variation of the same word. The rationale for this is that all the variants really refer to the same concept. There is no need for singular and plural. Many verbs can be stored in their stem form. Extending the concept, we can also map synonyms to the same token. Of course, this adds a layer of complexity to the processing of text. The gains in predictive performance are relatively modest, but the dictionary size will obviously be reduced. Stemming can occasionally be harmful for some words. If we apply a universal procedure that effectively trims words to their root form, we will encounter occasions where a subtle difference in meaning is missed. The words “exit” and “exiting” may appear to have identical roots, but in the context of programming and error messages, they may have different meanings. Overall, stemming will achieve a large reduction in dictionary size and is modestly beneficial for predictive performance when using a smaller dictionary.

In general, the smaller the dictionary, the more intelligence in its composition is needed to capture the most and best words. The use of tokens and stemming are examples of helpful procedures in composing smaller dictionaries. All these efforts will pay off in improved manageability of learning and perhaps improved accuracy. If nothing else is gained, learning can proceed more rapidly with smaller dictionaries.

Once the set of features has been determined, the document collection can be converted to spreadsheet format. Figure 2.5 shows an example of how this can be done for binary features. An implementation of a similar algorithm is available in the accompanying software. Each column in the spreadsheet corresponds to a feature. For interpretability, we will need to keep the list of features to translate from column number to feature name. And, of course, we will still need the document collection to be able to refer back to the original documents from the rows.

We have presented a model of data for predictive text mining in terms of a spreadsheet that is populated by ones or zeros. These cells represent the presence of the dictionary’s words in a document collection. To achieve the best predictive accuracy, we might consider additional transformations from this representation. Table 2.2 lists three different transformations that may improve predictive capabilities.

Word pairs and collocations are simple examples of multiword features discussed in more detail in Sect. 2.5.1. They serve to increase the size of the dictionary but can improve predictive performance in certain scenarios.

Fig. 2.5 Converting documents to a spreadsheet

```
Input:
   fs, a set of k features
   dc, a collection of n documents
Output: ss, a spreadsheet with n rows and k columns
Initialize: i := 1

for each document d in dc, do
   j := 1
   for each feature f in fs, do
      m := number of occurrences of f in d
      if (m > 0) then ss(row=i, col=j) := 1;
      else ss(row=i, col=j) := 0;
      endif
      increment j by 1
   endfor
   increment i by 1
endfor
output ss
```

Table 2.2 Dictionary feature transformations

Word pairs, collocations
Frequencies
tf-idf

Instead of zeros or ones as entries in the cells of the spreadsheet, the actual frequency of occurrence of the word could be used. If a word occurs ten times in a document, this count would be entered in the cell. We have all the information of a binary representation, and we have some additional information to contrast with other documents. For some learning methods, the count does give a slightly better result. It also may lead to more compact solutions because it includes the same solution space as the binary data model, yet the additional frequency information may yield a simpler solution. This is especially true of some learning methods whose solutions use only a small subset of the dictionary words. Overall, the frequencies are helpful in prediction but add complexity to the proposed solutions. One compromise that works quite well is to have a three-valued system for cell entries: a one or zero as in the binary representation, with the additional possibility of a 2. Table 2.3 lists the three possibilities, where we map all occurrences of more than two times into a maximum value of 2. Such a scheme seems to capture much of the added value of frequency information without adding much complexity to the model. Another variant involves zeroing values below a certain threshold on the plausible grounds that tokens should have a minimum frequency before being considered of any use. This can reduce the complexity of the spreadsheet significantly and might be a necessity for some data-mining algorithms. Besides simple thresholding, there are a variety of more sophisticated methods to reduce the spreadsheet complexity such as the use of chi-square, mutual information, odds ratio and others. Mutual information can be helpful when considering multiword features.

Table 2.3 Thresholding frequencies to three values

0—word did not occur
1—word occurred once
2—word occurred 2 or more times

The next step beyond counting the frequency of a word in a document is to modify the count by the perceived importance of that word. The well-known *tf-idf* formulation has been used to compute weightings or scores for words. Once again, the values will be positive numbers so that we capture the presence or absence of the word in a document. In (2.1), we see that the tf-idf weight assigned to word j is the term frequency (i.e., the word count) modified by a scale factor for the importance of the word. The scale factor is called the *inverse document frequency*, which is given in (2.2). It simply checks the number of documents containing word j (i.e., $\text{df}(j)$) and reverses the scaling. Thus, when a word appears in many documents, it is considered unimportant and the scale is lowered, perhaps near zero. When the word is relatively unique and appears in few documents, the scale factor zooms upward because it appears important.

$$\text{tf-idf}(j) = \text{tf}(j) * \text{idf}(j), \tag{2.1}$$

$$\text{idf}(j) = \log\left(\frac{N}{\text{df}(j)}\right). \tag{2.2}$$

Alternative versions of the basic tf-idf formulation exist, but the general motivation is the same. The net result of this process is a positive score that replaces the simple frequency or binary true-or-false entry in the cell of our spreadsheet. The bigger the score, the more important its expected value to the learning method. Although this transformation is only a slight modification of our original binary-feature model, it does lose the clarity and simplicity of the earlier presentation.

Another variant is to weight the tokens from different parts of the document differently. For example, the words in the subject line of a document could receive additional weight. An effective variant is to generate separate sets of features for the categories (for each category, the set of features is derived only from the tokens of documents of that category) and then pool all the feature sets together.

All of these models of data are modest variations of the basic binary model for the presence or absence of words. Which of the data transformations are best? We will not give a universal answer. Experience has shown that the best prediction accuracy is dependent on mating one of these variations to a specific learning method. The best variation for one method may not be the one for another method. Is it necessary to test all variations with all methods? When we describe the learning methods, we will give guidelines for the individual methods based on general research experience. Moreover, some methods have a natural relationship to one of these representations, and that alone would make them the preferred approach to representing data.

Much effort has been expended in transforming this word model of data into a somewhat more cryptic presentation. The data remain entries in the spreadsheet

Fig. 2.6 Spreadsheet to sparse vectors

Spreadsheet				Sparse Vectors
0	15	0	3	(2,15) (4,3)
12	0	0	0	(1,12)
8	0	5	2	(1,8) (3,5) (4,2)

cells, but their value may be less intelligible. Some of these transformations are techniques for reducing duplication and dimensions. Others are based on careful empirical experimentation that supports their value in increased predictive capabilities. We will discuss several classes of prediction methods. They tend to work better with different types of data transformations.

Although we describe data as populating a spreadsheet, we expect that most of the cells will be zero. Most documents contain a small subset of the dictionary's words. In the case of text classification, a text corpus might have thousands of word types. Each individual document, however, has only a few hundred unique tokens. So, in the spreadsheet, almost all of the entries for that document will be zero. Rather than store all the zeros, it is better to represent the spreadsheet as a set of *sparse vectors*, where a row is represented by a list of pairs, one element of the pair being a column number and the other element being the corresponding nonzero feature value. By not storing the zeros, savings in memory can be immense. Processing programs can be easily adapted to handle this format. Figure 2.6 gives a simple example of how a spreadsheet is transformed into sparse vectors. All of our proposed data representations are consistent with such a sparse data representation.

2.5.1 Multiword Features

Generally, features are associated with single words (tokens delimited by white space). Although this is reasonable most of the time, there are cases where it helps to consider a group of words as a feature. This happens when a number of words are used to describe a concept that must be made into a feature. The simplest scenario is where the feature space is extended to include pairs of words. Instead of just separate features for *bon* and *vivant*, we could also have a feature for *bon vivant*. But why stop at pairs? Why not consider more general *multiword* features?

The most common example of this is a named entity, for example *Don Smith* or *United States of America*. Unlike word pairs, the words need not necessarily be consecutive. For example, in specifying *Don Smith* as a feature, we may want to ignore the fact that he has a middle name of *Leroy* that may appear in some references to the person. Another example of a multiword feature is an adjective followed by a noun, such as *broken vase*. In this case, to accommodate many references to the

noun that involve a number of adjectives with the desired adjective not necessarily adjacent to the noun, we must permit some flexibility in the distance between the adjective and noun. In the same example of the vase, we want to accept a phrase such as *broken and dirty vase* as an instance of *broken vase*. An even more abstract case is when words simply happen to be highly correlated in the text. For instance, in stories about Germany boycotting product Y, the word-stem *German* would be highly correlated with the wordstem *boycott* within a small window (say, five words). Thus, more generally, multiword features consist of x number of words occurring within a maximum window size of y (with $y \geq x$ naturally).

The key question is how such features can be extracted from text. How smart do we have to be in finding such features? Named entities can be extracted using specialized methods. For other multiword features, a more general approach might be to treat them like single-word features. If we use a frequency approach, then we will only include those combinations of words that occur relatively frequently. A straightforward implementation would simply examine all combinations of up to x words within a window of y words. Clearly, the number of potential features grows significantly when multiword features are considered.

Measuring the value of multiword features is typically done by considering correlation between the words in potential multiword features. A variety of measures based on mutual information or the likelihood ratio may be used for this purpose. In the accompanying software, (2.3), which computes an association measure AM for the multiword T, is used for evaluating multiword features, where size(T) is the number of words in phrase T and freq(T) is the number of times phrase T occurs in the document collection.

$$AM(T) = \frac{\text{size}(T)\log_{10}(\text{freq}(T))\text{freq}(T)}{\sum_{word_i \in T}\text{freq}(word_i)}. \tag{2.3}$$

Other variations can occur depending on whether stopwords are excluded before building multiword features.

An algorithm for generating multiword features is shown in Fig. 2.7, which extends Fig. 2.4 to multiword features. A straightforward implementation can consume a lot of memory, but a more efficient implementation uses a sliding window to generate potential multiwords in a single pass over the input text without having to store too many words in memory. A version of this is implemented in the accompanying software.

Generally, multiword features are not found too frequently in a document collection, but when they do occur they are often highly predictive. They are also particularly satisfying for explaining a learning method's proposed solution. The downside to using multiwords is that they add an additional layer of complexity to the processing of text, and some practitioners may feel it's the job of the learning methods to combine the words without a preprocessing step to compose multiword features. However, if the learning method is not capable of doing this, the extra effort may be worthwhile because multiwords are often highly predictive and enhance the interpretability of results.

```
Input:
   ts, sequence of tokens in the document collection
   k, the number of features desired
   mwl, maximum length of multiword
   mws, maximum span of words in multiword
   slvl, correlation threshold for multiword features
   mfreq, frequency threshold for accepting features
Output:
   fs, a set of k features
Initialize:
   hs := empty hashtable

for each tok in ts do
   Generate a list of multiword tokens ending in tok.
   This list includes the single-word tok and uses the inputs mws and mwl.
   Call this list mlist.
   for each mtok in mlist do
      If hs contains mtok then
         i := value of mtok in hs
         increment i by 1
      else
         i := 1
      endif
      store i as value of mtok in hs
   endfor
endfor
sk := keys in hs sorted by decreasing value
delete elements in sk with a frequency < mfreq
delete multiword elements in sk with an association measure < slvl
fs := top k keys in sk
output fs
```

Fig. 2.7 Generating multiword features from tokens

2.5.2 Labels for the Right Answers

For prediction, an extra column must be added to the spreadsheet. This last column of the spreadsheet, containing the label, looks no different from the others. It is a one or zero indicating that the correct answer is either true or false. What is the label? Traditionally, this label has been a topic to index the document. Sports or financial stories are examples of topics. We are not making this semantic distinction. Any answer that can be measured as true or false is acceptable. It could be a topic or category, or it could be an article that appeared prior to a stock price's rise. As long as the answers are labeled correctly relative to the concept, the format is acceptable. Of course, that doesn't mean that the problem can readily be solved. In the sparse vector format, the labels are appended to each vector separately as either a one (positive class) or a zero (negative class).

2.5.3 Feature Selection by Attribute Ranking

In addition to the frequency-based approaches mentioned earlier, feature selection can be done in a number of different ways. In general, we want to select a set of features for each category to form a local dictionary for the category. A relatively simple and quite useful method for doing so is by independently ranking feature attributes according to their predictive abilities for the category under consideration. In this approach, we can simply select the top-ranking features.

The predictive ability of an attribute can be measured by a certain quantity that indicates how correlated a feature is with the class label. Assume that we have n documents, and x_j is the presence or absence of attribute j in a document x. We also use y to denote the label of the document; that is, the last column in our spreadsheet model. A commonly used ranking score is the *information gain* criterion, which can be defined as

$$IG(j) = L_{label} - L(j),$$

where

$$L_{label} = \sum_{c=0}^{1} \Pr(y = c) \log_2 \frac{1}{\Pr(y = c)}, \tag{2.4}$$

$$L(j) = \sum_{v=0}^{1} \Pr(x_j = v) \sum_{c=0}^{1} \Pr(y = c | x_j = v) \log_2 \frac{1}{\Pr(y = c | x_j = v)}. \tag{2.5}$$

The quantity $L(j)$ is the number of bits required to encode the label and the attribute j minus the number of bits required to encode the attribute j. That is, $L(j)$ is the number of bits needed to encode the label given that we know the attribute j. Therefore, the information gain $L_{label} - L(j)$ is the number of bits we can save for encoding the class label if we know the feature j. Clearly, it measures how useful a feature j is from the information-theoretical point of view.

Since L_{label} is the same for all j, we can simply compute $L(j)$ for all attributes j and select the ones with the smallest values. Quantities that are needed to compute $L(j)$ in (2.5) can be easily estimated using the following plug-in estimators:

$$\Pr(x_j = v) = \frac{\text{freq}(x_j = v) + 1}{n + 2},$$

$$\Pr(y = c | x_j = v) = \frac{\text{freq}(x_j = v, label = c) + 1}{\text{freq}(x_j = v) + 2}.$$

2.6 Sentence Boundary Determination

If the XML markup for a corpus does not mark sentence boundaries, it is often necessary for these to be marked. At the very least, it is necessary to determine when a

period is part of a token and when it is not. For more sophisticated linguistic parsing, the algorithms often require a complete sentence as input. We shall also see other information extraction algorithms that operate on text a sentence at a time. For these algorithms to perform optimally, the sentences must be identified correctly. Sentence boundary determination is essentially the problem of deciding which instances of a period followed by whitespace are sentence delimiters and which are not since we assume that the characters ? and ! are unambiguous sentence boundaries. Since this is a classification problem, one can naturally invoke standard classification software on training data and achieve accuracy of more than 98%. This is discussed at some length in Sect. 2.12. However, if training data are not available, one can use a hand-crafted algorithm.

Figure 2.8 gives an algorithm that will achieve an accuracy of more than 90% on newswire text. Adjustments to the algorithm for other corpora may be necessary to get better performance. Notice how the algorithm is implicitly tailored for English. A different language would have a completely different procedure but would still involve the basic idea of rules that examine the context of potential sentence boundaries. A more thorough implementation of this algorithm is available in the accompanying software.

Input: a text with periods
Output: same text with End-of-Sentence (EOS) periods identified

Overall Strategy:
1. Replace all identifiable non-EOS periods with another character
2. Apply rules to all the periods in text and mark EOS periods
3. Retransform the characters in step 1 to non-EOS periods
4. Now the text has all EOS periods clearly identified

Rules:
All ? ! are EOS
If " or ’ appears before period, it is EOS
If the following character is not white space, it is not EOS
If) }] before period, it is EOS
If the token to which the period is attached is capitalized
and is < 5 characters and the next token begins uppercase,
it is not EOS
If the token to which the period is attached has other periods,
it is not EOS
If the token to which the period is attached begins with a lowercase
letter and the next token following whitespace is uppercase,
it is EOS
If the token to which the period is attached has < 2 characters,
it is not EOS
If the next token following whitespace begins with $ ({ [" ’ it is EOS
Otherwise, the period is not EOS

Fig. 2.8 End-of-sentence detection algorithm

2.7 Part-of-Speech Tagging

Once a text has been broken into tokens and sentences, the next step depends on what is to be done with the text. If no further linguistic analysis is necessary, one might proceed directly to feature generation, in which the features will be obtained from the tokens. However, if the goal is more specific, say recognizing names of people, places, and organizations, it is usually desirable to perform additional linguistic analyses of the text and extract more sophisticated features. Toward this end, the next logical step is to determine the *part of speech* (POS) of each token.

In any natural language, words are organized into grammatical classes or parts of speech. Almost all languages will have at least the categories we would call nouns and verbs. The exact number of categories in a given language is not something intrinsic but depends on how the language is analyzed by an individual linguist.

In English, some analyses may use as few as six or seven categories and others nearly one hundred. Most English grammars would have as a minimum noun, verb, adjective, adverb, preposition, and conjunction. A bigger set of 36 categories is used in the Penn Tree Bank, constructed from the Wall Street Journal corpus discussed later on. Table 2.4 shows some of these categories.

Table 2.4 Some of the categories in the penn tree bank POS set

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
POS	Possessive ending
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
WDT	Wh-determiner

Dictionaries showing word—POS correspondence can be useful but are not sufficient. All dictionaries have gaps, but even for words found in the dictionary, several parts of speech are usually possible. Returning to an earlier example, "bore" could be a noun, a present tense verb, or a past tense verb. The goal of POS tagging is to determine which of these possibilities is realized in a particular text instance.

Although it is possible, in principle, to manually construct a part-of-speech tagger, the most successful systems are generated automatically by machine-learning algorithms from annotated corpora. Almost all POS taggers have been trained on the Wall Street Journal corpus available from LDC (Linguistic Data Corporation, www.ldc.upenn.edu) because it is the most easily available large annotated corpus. Although the WSJ corpus is large and reasonably diverse, it is one particular genre, and one cannot assume that a tagger based on the WSJ will perform as well on, for example, e-mail messages. Because much of the impetus for work on information extraction has been sponsored by the military, whose interest is largely in the processing of voluminous news sources, there has not been much support for generating large training corpora in other domains.

2.8 Word Sense Disambiguation

English words, besides being ambiguous when isolated from their POS status, are also very often ambiguous as to their meaning or reference. Returning once again to the example "bore," one cannot tell without context, even after POS tagging, if the word is referring to a person—"he is a bore"—or a reference to a hole, as in "the bore is not large enough." The main function of ordinary dictionaries is to catalog the various meanings of a word, but they are not organized for use by a computer program for disambiguation. A large, long-running project that focused on word meanings and their interrelationships is Wordnet, which aimed to fill in this gap. As useful as Wordnet is, by itself it does not provide an algorithm for selecting a particular meaning for a word in context. In spite of substantial work over a long period of time, there are no algorithms that can completely disambiguate a text. In large part, this is due to the lack of a huge corpus of disambiguated text to serve as a training corpus for machine-learning algorithms. Available corpora focus on relatively few words, with the aim of testing the efficacy of particular procedures. Unless a particular text-mining project can be shown to require word sense disambiguation, it is best to proceed without such a step.

2.9 Phrase Recognition

Once the tokens of a sentence have been assigned POS tags, the next step is to group individual tokens into units, generally called phrases. This is useful both for creating a "partial parse" of a sentence and as a step in identifying the "named entities" occurring in a sentence, a topic we will return to in greater detail later

on. There are standard corpora and test sets for developing and evaluating phrase recognition systems that were developed for various research workshops. Systems are supposed to scan a text and mark the beginnings and ends of phrases, of which the most important are noun phrases, verb phrases, and prepositional phrases. There are a number of conventions for marking, but the most common is to mark a word inside a phrase with I-, a word at the beginning of a phrase adjacent to another phrase with B- and a word outside any phrase with O. The I- and B- tags can be extended with a code for the phrase type: I-NP, B-NP, I-VP, B-VP, etc. Formulated in this way, the phrase identification problem is reduced to a classification problem for the tokens of a sentence, in which the procedure must supply the correct class for each token.

Performance varies widely over phrase type, although overall performance measures on benchmark test sets are quite good. A simple statistical approach to recognizing significant phrases might be to consider multiword tokens. If a particular sequence of words occurs frequently enough in the corpora, it will be identified as a useful token.

2.10 Named Entity Recognition

A specialization of phrase finding, in particular noun phrase finding, is the recognition of particular types of proper noun phrases, specifically persons, organizations, and locations, sometimes along with money, dates, times, and percentages. The importance of these recognizers for intelligence applications is easy to see, but they have more mundane uses, particularly in turning verbose text data into a more compact structural form.

From the point of view of technique, this is very like the phrase recognition problem. One might even want to identify noun phrases as a first step. The same sort of token-encoding pattern can be used (B-person, B-location, I-person, etc.), and the problem is then one of assigning the correct class code to each token in a sentence. We shall discuss this problem in detail in Chap. 6 on information extraction.

2.11 Parsing

The most sophisticated kind of text processing we will consider, briefly, is the step of producing a full parse of a sentence. By this, we mean that each word in a sentence is connected to a single structure, usually a tree but sometimes a directed acyclic graph. From the parse, we can find the relation of each word in a sentence to all the others, and typically also its function in the sentence (e.g. subject, object, etc.). There are very many different kinds of parses, each associated with a linguistic theory of language. This is not the place to discuss these various theories. For our purposes, we can restrict attention to the so-called “context-free” parses. One can envision a parse of this kind as a tree of nodes in which the leaf nodes are the words of a sentence, the phrases into which the words are grouped are internal nodes, and

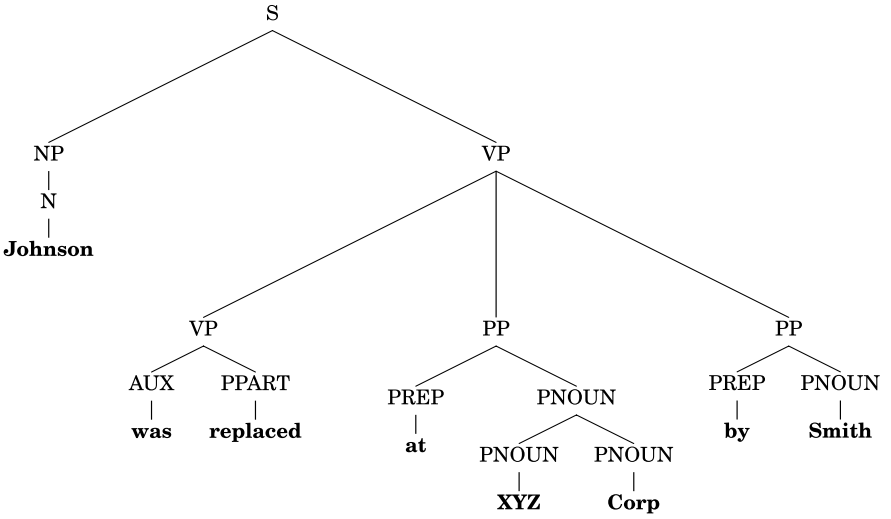

Fig. 2.9 Simple parse tree

```
 Johnson was replaced at XYZ Corp. by Smith .
-----------------------------------------------------------------------
.----- subj(n)      Johnson1(1)         noun propn sg h m gname sname
o----- top          be(2,1,3)           verb vfin vpast sg vsubj
'----- pred(en)     replace1(3,7,1,u)   verb ven vpass
  '--- vprep        at1(4,6)            prep pprefv staticp
  | '- objprep(n)   XYZ Corp.1(6)       noun propn sg glom ctitle
  '--- subj(agent)  by1(7,8)            prep pprefv
    '- objprep(n)   Smith1(8)           noun propn sg h sname
```

Fig. 2.10 Parse tree—English Slot Grammar

there is one top node at the root of the tree, which usually has the label S. There are a number of algorithms for producing such a tree from the words of a sentence. Considerable research has been done on constructing parsers from a statistical analysis of tree banks of sentences parsed by hand. The best-known and most widely used tree bank is of parsed sentences from the Wall Street Journal and is available from LDC.

The reason for considering such a comparatively expensive process is that it provides information that phrase identification or partial parsing cannot provide. Consider a sentence such as "Johnson was replaced at XYZ Corp. by Smith." for which a simple parse is shown in Fig. 2.9.

From the linear order of phrases in a partial parse, one might conclude that Johnson replaced Smith. A parse that identifies the sentence as passive has information allowing the extraction of the correct "Smith replaced Johnson." An example of a parse giving more information than a simple tree is shown in Fig. 2.10. The parse

is the output of the English Slot Grammar. In this example, the tree is drawn, using printer characters, on its side, with the top of the tree to the left. Notice in particular that the "by" phrase containing "Smith" is identified as the agent, although "Johnson" is marked as the subject.

2.12 Feature Generation

Although we emphasized that our focus is on statistical methods, the reason for the linguistic processing described earlier is to identify features that can be useful for text mining. As an example, we will consider how good features depend on the type of object to be classified and how such features can be obtained using the processes discussed so far.

Let us consider the problem of segmenting a text into sentences. A hand-crafted algorithm for this task was given earlier. However, let's say we want to learn a set of similar rules from training data instead. What sorts of features would we generate? Since the object to be classified is a period, each feature vector corresponds to a period occurring in the text. Now we need to consider what characteristics of the surrounding text are useful features. From the algorithm, we can see that the useful features are the characters or character classes near the period, including the character of the token to which the period is attached and the characters of the following token. Therefore, the necessary linguistic processing only involves tokenization of the text.

A more sophisticated example would be the identification of the part of speech (POS) of each word in a text. This is normally done on a text that has first been segmented into sentences. The influence of one word on the part-of-speech of another does not cross sentence boundaries. The object that a feature vector represents is a token. Features that might be useful in identifying the POS are, for example, whether or not the first letter is capitalized (marking a proper noun), if all the characters are digits, periods, or commas (marking a number), if the characters are alternating uppercase letters and periods (an abbreviation), and so on. We might have information from a dictionary as to the possible parts of speech for a token. If we assume the POS assignment goes left to right through a sentence, we have POS assignments of tokens to the left as possible features. In any case, we have the identity of tokens to the left and to the right. For example, "the" most likely precedes either a noun or an adjective. So, for this task, we basically need tokenization plus analysis of the tokens, plus perhaps some dictionaries that tell what the possibilities are for each token in order to create a feature vector for each token.

The feature vector for a document is assigned a particular class (or set of classes). The feature vector for classifying periods as End-Of-Sentence or not is assigned to one of two classes. The feature vector for POS assignment has one of a finite set of classes. The class of the feature vector for each token in the partial parsing task was outlined above. These classes are not intrinsic or commonly agreed properties of a token. They are invented constructs specific to the problem. Let us consider what features are important for the feature vector for partial parsing. Token identity

is clearly one of these, as is the token POS. Additionally, the identity and POS of the tokens to the left and right of the token whose vector is being constructed are important features. So is the phrasal class of tokens to the left that have already been assigned. Sentence boundaries are particularly important since phrases do not cross them. Individual token features, on the other hand, are not important because they have already been taken into account for POS assignment.

For named entity detection, the same kind of token class encoding scheme can be used as in the chunking task (i.e., B-Person, I-Person, etc.). All the named entities are noun phrases, so it is possible but not necessary that a sentence will first be segmented into noun phrases. This might result in unnecessary work since named entities are typically made up of proper nouns. For this task, dictionaries can be particularly important. One can identify tokens as instances of titles, such as "Doctor" or "President," providing clues as to the class of proper noun phrases to the right. Other dictionaries can list words that are typically the end of organization names, like "Company," "Inc.," or "Department." There are also widely available gazetteers (i.e., lists of place names and lists of organization names). Identifying a token as being a piece of such a dictionary entry is useful but not definitive because of ambiguity of names (e.g., "Arthur Anderson" might be referring to a person or to a company). Besides the dictionary information, other useful features are POS, sentence boundaries, and the class of tokens already assigned.

2.13 Summary

Documents are composed of words, and machine learning methods process numerical vectors. This chapter discusses how words are transformed into vectors, readying them for processing by predictive methods. Documents may appear in different formats and may be collected from different sources. With minor modifications, they can be organized and unified for prediction by specifying them in a standard descriptive language, XML. The words or tokens may be further reduced to common roots by stemming. These tokens are added to a dictionary. The words in a document can be converted to vectors using local or global dictionaries. The value of each entry in the vector will be based on measures of frequency of occurrence of words in a document such as term frequency (tf and idf). An additional entry in a document vector is a label of the correct answer, such as its topic. Dictionaries can be extended to multiword features like phrases. Dictionary size may be significantly reduced by attribute ranking. The general approach is purely empirical, preparing data for statistical prediction. Linguistic concepts are also discussed including part-of-speech tagging, word sense disambiguation, phrase recognition, parsing and feature generation.

2.14 Historical and Bibliographical Remarks

A detailed account of linguistic processing issues can be found in Indurkhya and Damerau (2010) and Jurafsky and Martin (2008). Current URLs for organizations

such as LDC, ICAME, TEI, the Oxford Text Archive, and the like are easily found through a search engine. The new Reuters corpus, RCV1, is discussed in Lewis *et al.* (2004) and is available directly from Reuters. The older Reuters-21578 Distribution 1.0 corpus is also available on the Internet at several sites. Using a search engine with the query "download Reuters 21578" will provide a list of a number of sites where this corpus can be obtained. There are a number of Web sites that have many links to corpora in many languages. Again, use of a search engine with the query "corpus linguistics" will give the URLs of active sites. There are many books on XML; for example Ray (2001). The best-known algorithm for derivational regularization is the Porter stemmer (Porter 1980), which is in the public domain. At the time of this writing, it can be downloaded from http://www.tartarus.org/~martin/PorterStemmer. ANSI C, Java, and Perl versions are available. Another variation on stemming is to base the unification of tokens into stems on corpus statistics (i.e., the stemming is corpus based) (Xu and Croft 1998). For information retrieval, this algorithm is said to provide better results than the more aggressive Porter stemmer.

For end-of-sentence determination, Walker *et al.* (2001) compares a hard-coded program, a rule-based system, and a machine-learning solution on the periods and some other characters in a collection of documents from the Web. The machine-learning system was best of the three. The F-measures (see Sect. 3.5.1) were 98.37 for the machine-learning system, 95.82 for the rule-based system, and 92.45 for the program. Adjusting a machine-learning solution to a new corpus is discussed in Zhang *et al.* (2003).

Examples of part-of-speech taggers are Ratnaparkhi (1995) and Brill (1995). The Brill tagger is in the public domain and is in wide use.

For a survey of work on word sense disambiguation, see Ide and Véronis (1998). Wordnet is discussed in Feldbaum (1998). The database and a program to use it can be obtained from the Internet at http://wordnet.princeton.edu.

Phrase recognition is also known as "text chunking" (Sang and Buchholz 2000). A number of researchers have investigated this problem as classification, beginning with Ramshaw and Marcus (1995). A variety of machine-learning algorithms have been used: support vector machines (Kudoh and Matsumoto 2000), transformation-based learning (Ramshaw and Marcus 1995), a linear classifier (Zhang *et al.* 2002), and others. Many more details can be found at http://ifarm.nl/signll/conll.

Work on named entity recognition has been heavily funded by the US government, beginning with the Message Understanding Conferences and continuing with the ACE project Further details can be obtained from the following sites:

http://www.itl.nist.gov/iaui/894.02/related_projects/muc/index.html
http://www.itl.nist.gov/iaui/894.01/tests/ace/index.html.

A number of named entity recognition systems are available for license, such as the Nymble system from BBN (Bikel *et al.* 1997). State of the art is about 90% in recognition accuracy.

Algorithms for constructing context-free parse trees are discussed in Earley (1970) and Tomita (1985). Constructing parsers from tree-banks is discussed in

Charniak (1997), Ratnaparkhi (1999), Chiang (2000), and others. A description of English Slot Grammar can be found in McCord (1989). A number of organizations have posted demo versions of their parsers on Web sites. These can be used to compare the output of different parsers on the same input sentence. Three examples of such sites are:

http://www.agfl.cs.ru.nl/EP4IR/english.html
http://www.link.cs.cmu.edu/link/submit-sentence-4.html
http://www2.lingsoft.fi/cgi-bin/engcg.

Early work in feature generation for document classification includes (Lewis 1992; Apté *et al.* 1994), and others. Tan *et al.* (2002) showed that bigrams plus single words improved categorization performance in a collection of Web pages from the Yahoo Science groups. The better performance was attributed to an increase in recall. A special issue of the *Journal of Machine Learning Research*, in 2003 was devoted to feature selection and is available online. One of the papers (Forman 2003) presents experiments on various methods for feature reduction. A useful reference on word selection methods for dimensionality reduction is Yang and Pedersen (1997), which discusses a wide variety of methods for selecting words useful in categorization. It concludes that document frequency is comparable in performance to expensive methods such as information gain or chi-square.

2.15 Questions and Exercises

1. How would you evaluate a tokenizer?
2. What component in the software set would need to be replaced in order to run a classifier for periods as end-of-sentence?
3. Create a properties file for mkdict for Reuters-21578 train data set with no special parameter settings and run using a size of 500. Don't forget to set all the tags (especially bodytags and doctag) in the properties file and ensure that it is in the current directory where the java command is executed.
4. Modify the parameters in the properties file to use stop words and stemming. You can use the files provided in ExerciseFiles.zip that you downloaded earlier. Run and note the differences in the dictionary from the previous exercise.
5. Generate a local dictionary for the category *earn*. This dictionary will be built from documents that have the topic *earn*. Check the dictionary file. Lets say you don't want any of the numeric features or ones that include the characters #, & and -. You could edit the file and delete these manually. Or else you could edit the properties file and regenerate the local dictionary. (Hint: add the characters you want to exclude to BOTH whitespace-chars and word-delimiters.) Note that the dictionary is still not perfect, but we will let the prediction programs decide which features to use.
6. Use the dictionary from the previous exercise to generate vectors for the category *earn* for both the training and test data sets.

http://www.springer.com/978-1-84996-225-4

Fundamentals of Predictive Text Mining
Weiss, S.M.; Indurkhya, N.; Zhang, T.
2010, XIV, 226 p., Hardcover
ISBN: 978-1-84996-225-4