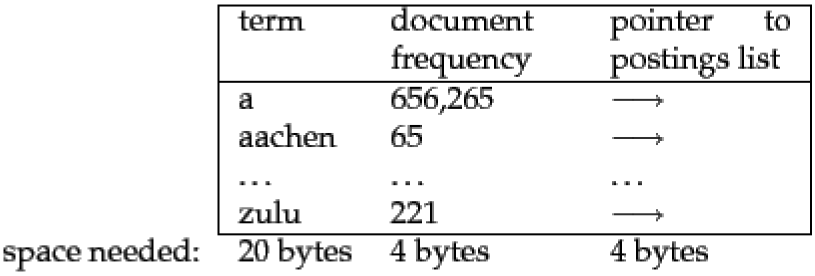
<u>**UNIT 3**</u>

Compression

Dictionary compression, Postings file compression, Variable byte codes, Gamma codes. Vector Space Model- Parametric and zone indexes, Learning weights, Term frequency and weighting, Tf-Idf weighting, Vector space model for scoring, variant tf-idf functions.

Dictionary compression

The search system on your PC must get along with the memory-hogging word processing suite you are using at the same time.



	term	document frequency	pointer to postings list
	a	656,265	⟶
	aachen	65	⟶
	…	…	…
	zulu	221	⟶
space needed:	20 bytes	4 bytes	4 bytes

Figure 5.3: Storing the dictionary as an array of fixed-width entries.

Dictionary as a string

. For large collections like the web, we need to allocate more bytes per pointer. We look up terms in the array by binary search.

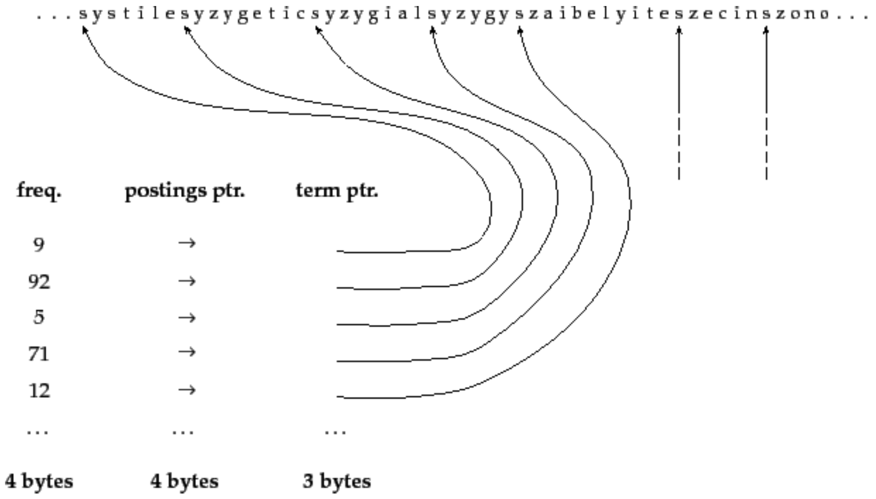
For Reuters-RCV1, we

$$M \times (2 \times 20 + 4 + 4) = 400{,}000 \times 48 = 19.2 \text{ megabytes (MB)}$$

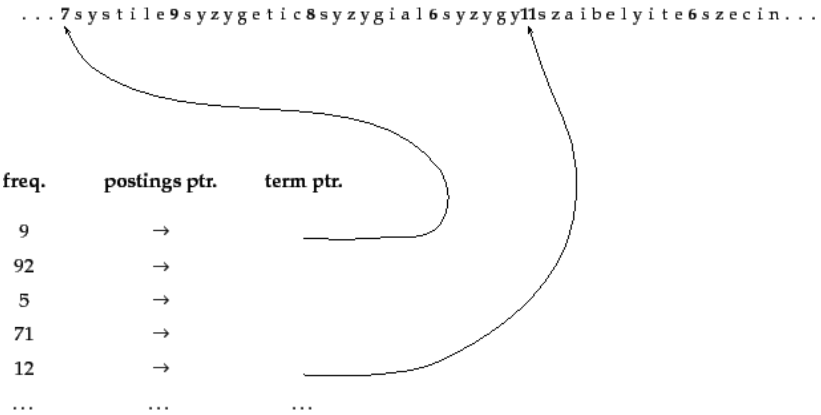
need

$$M \times (20 + 4 + 4) = 400{,}000 \times 28 = 11.2\text{megabytes (MB)}$$

for storing the dictionary in this scheme.


. . . s y s t i l e s y z y g e t i c s y z y g i a l s y z y g y s z a i b e l y i t e s z e c i n s z o n o . . .
freq.
postings ptr.
term ptr.
9
92
5
71
12
…
→
→
→
→
→
…
…
4 bytes
4 bytes
3 bytes

Dictionary-as-a-string storage.Pointers mark the end of the preceding term and the beginning of the next.
For example, the first three terms in this example are systile, syzygetic, and syzygial.


. . . 7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin. . .
freq.
postings ptr.
term ptr.
9
92
5
71
12
. . .
. . .
. . .

Blocked storage with four terms per block.The first block consists of systile, syzygetic, syzygial, and syzygy with lengths of seven, nine, eight, and six characters, respectively. Each term is preceded by a byte encoding its length that indicates how many bytes to skip to reach subsequent terms.

Blocked storage

We can further compress the dictionary by grouping terms in the string into of size k and keeping a term pointer only for the first term of each block (Figure 5.5). We store the length of the term in the string as an additional byte at the beginning of the term. We thus eliminate $k-1$ term pointers, but need an additional k bytes for storing the length of each term. For $k=4$, we save $(k-1)\times 3=9$ bytes for term pointers, but need an additional $k=4$ bytes for term lengths. So the total space requirements for the dictionary of Reuters-RCV1 are reduced by 5 bytes per four-term block, or a total of $400{,}000\times 1/4\times 5=0.5\ \mathrm{MB}$, bringing us down to 10.37.1 MB.

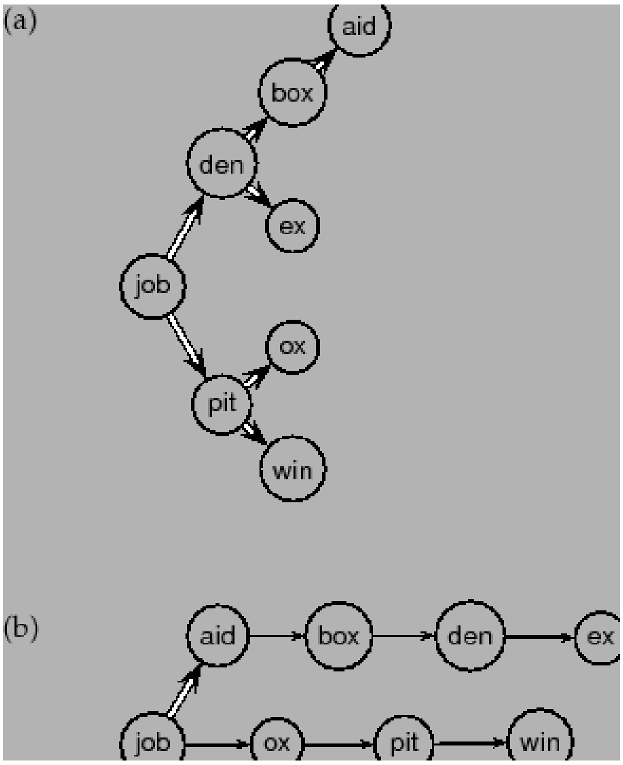


Figure 5.6: Search of the uncompressed dictionary (a) and a dictionary compressed by blocking with $k = 4$ (b).

Table 5.2: Dictionary compression for Reuters-RCV1.

	data structure	size in MB	
	dictionary, fixed-width	19.211.2	
	dictionary, term pointers into string	10.8 7.6	
	∼, with blocking, $k = 4$	10.3 7.1	
	∼, with blocking & front coding	7.9 5.9	

Postings file compression

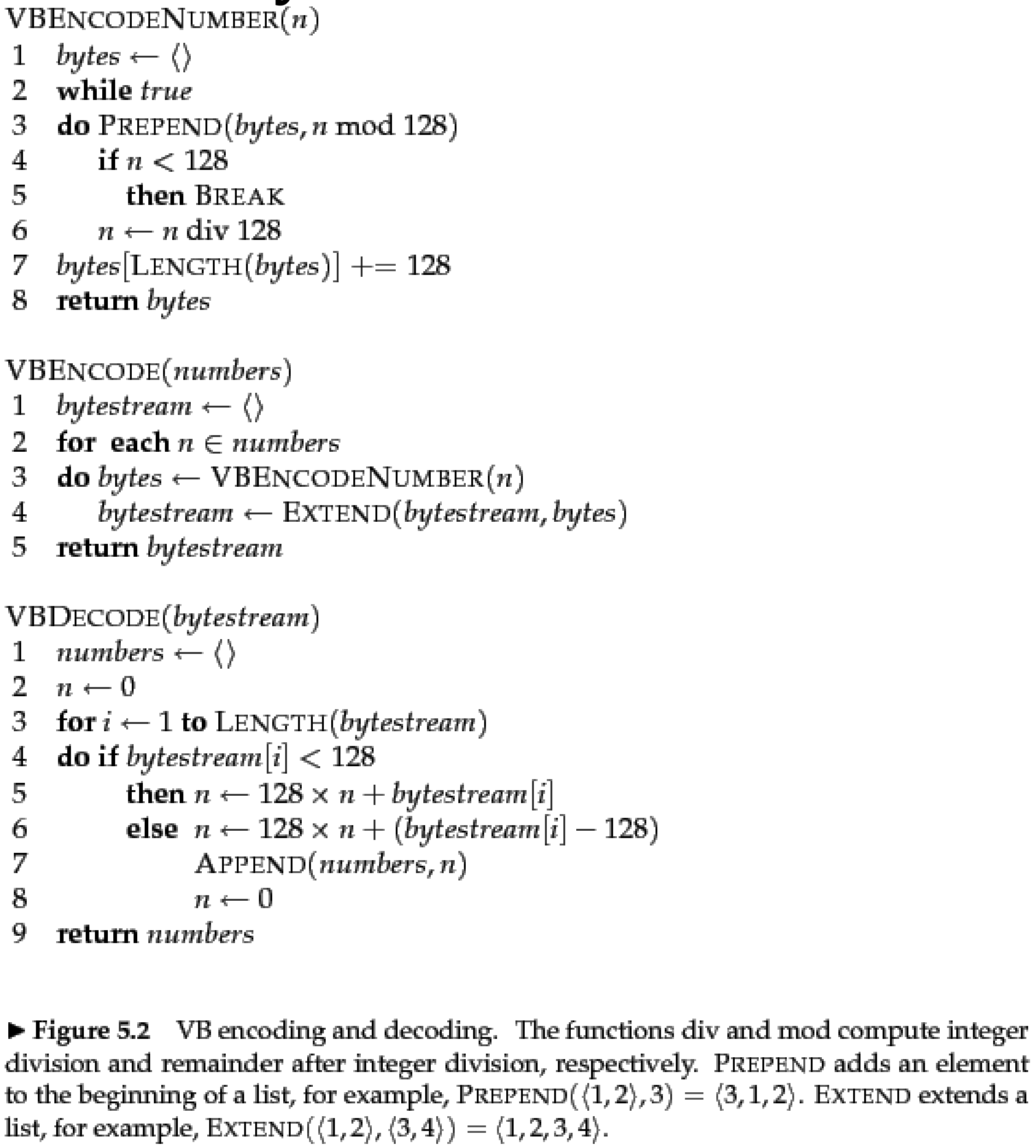
Table: Encoding gaps instead of document IDs. For example, we store gaps 107, 5, 43, ..., instead of docIDs 283154, 283159, 283202, ... for computer. The first docID is left unchanged (only shown for arachnocentric).

		encoding	postings list										
	the	docIDs	...		283042		283043		283044		283045	...	
		gaps				1		1		1		...	
	computer	docIDs	...		283047		283154		283159		283202	...	
		gaps				107		5		43		...	
	arachnocentric	docIDs	252000		500100								
		gaps	252000	248100									

Recall from Table 4.2 (page 4.2) that Reuters-RCV1 has 800,000 documents, 200 tokens per document, six characters per token, and 100,000,000 postings where we define a posting in this chapter as a docID in a postings list, that is, excluding frequency and position information. These numbers correspond to line 3 (``case folding'') in Table 5.1 . Document identifiers are $\log_2 800{,}000 \approx 20$ bits long. Thus, the size of the collection is about $800{,}000 \times 200 \times 6 \text{ bytes} = 960 \text{ MB}$ and the size of the uncompressed postings file is $100{,}000{,}000 \times 20/8 = 250 \text{ MB}$.

To encode small numbers in less space than large numbers, we look at two types of methods: bytewise compression and bitwise compression. As the names suggest, these methods attempt to encode gaps with the minimum number of bytes and bits, respectively

Variable byte codes



```
VBEncodeNumber($n$)
bytes $\leftarrow \langle\rangle$
while true
do Prepend(bytes, $n$ mod 128)
    if $n < 128$
       then Break
    $n \leftarrow n$ div 128
bytes[Length(bytes)] += 128
return bytes

VBEncode(numbers)
bytestream $\leftarrow \langle\rangle$
for each $n \in$ numbers
do bytes $\leftarrow$ VBEncodeNumber($n$)
    bytestream $\leftarrow$ Extend(bytestream, bytes)
return bytestream

VBDecode(bytestream)
numbers $\leftarrow \langle\rangle$
$n \leftarrow 0$
for $i \leftarrow 1$ to Length(bytestream)
do if bytestream[$i$] $< 128$
       then $n \leftarrow 128 \times n +$ bytestream[$i$]
       else $n \leftarrow 128 \times n +$ (bytestream[$i$] $- 128$)
            Append(numbers, $n$)
            $n \leftarrow 0$
return numbers
```

► **Figure 5.2** VB encoding and decoding. The functions div and mod compute integer division and remainder after integer division, respectively. Prepend adds an element to the beginning of a list, for example, Prepend($\langle 1,2\rangle$, 3) = $\langle 3,1,2\rangle$. Extend extends a list, for example, Extend($\langle 1,2\rangle, \langle 3,4\rangle$) = $\langle 1,2,3,4\rangle$.

VB encoding. Gaps are encoded using an integral number of bytes. The first bit, the continuation bit, of each byte indicates whether the code ends with this byte (1) or not (0).

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Gamma codes

Table 5.5: Some examples of unary and γ codes. Unary codes are only shown for the smaller numbers. Commas in γ codes are for readability only and are not part of the actual codes.

	number	unary code	length	offset	γ code	
	0	0				
	1	10	0		0	
	2	110	10	0	10,0	
	3	1110	10	1	10,1	
	4	11110	110	00	110,00	
	9	1111111110	1110	001	1110,001	
	13		1110	101	1110,101	
	24		11110	1000	11110,1000	
	511		111111110	11111111	111111110,11111111	
	1025		11111111110	0000000001	11111111110,0000000001	

VB codes

VB codes use an adaptive number of *bytes* depending on the size of the gap. Bit-level codes adapt the length of the code on the finer grained *bit* level. The simplest bit-level code is *unary code* . The unary code of n is a string of n 1s followed by a 0 (see the first two columns of Table 5.5). Obviously, this is not a very efficient code, but it will come in handy in a moment.

How efficient can a code be in principle? Assuming the 2^n gaps G with $1 \leq G \leq 2^n$ are all equally likely, the optimal encoding uses n bits for each G. So some gaps ($G = 2^n$ in this case) cannot be encoded with fewer than $\log_2 G$ bits. Our goal is to get as close to this lower bound as possible.

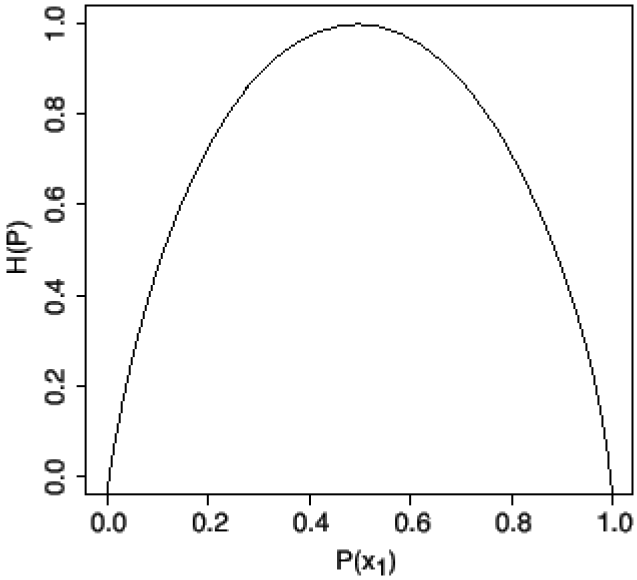


Figure 5.9: Entropy $H(P)$ as a function of $P(x_1)$ for a sample space with two outcomes x_1 and x_2 .

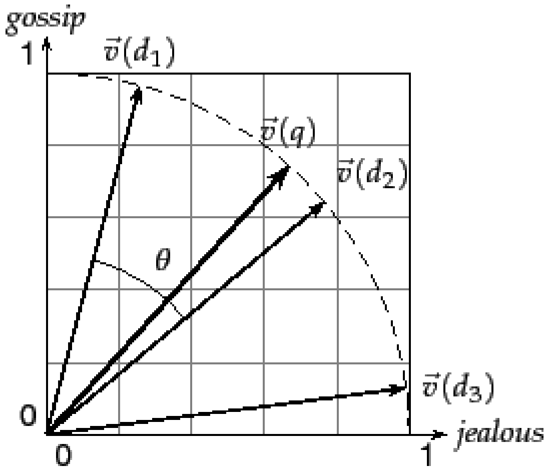
Table: Index and dictionary compression for Reuters-RCV1. The compression ratio depends on the proportion of actual text in the collection. Reuters-RCV1 contains a large amount of XML markup. Using the two best compression schemes, γ encoding and blocking with front coding, the ratio compressed index to collection size is therefore especially small for Reuters-RCV1: $(101 + 7.9)/3600 \approx 0.03$. $(101 + 5.9)/3600 \approx 0.03$.

	data structure	size in MB

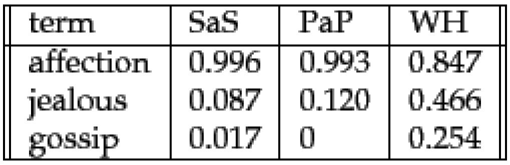
	dictionary, fixed-width	19.211.2
	dictionary, term pointers into string	10.8 7.6
	$\sim$, with blocking, $k = 4$	10.3 7.1
	$\sim$, with blocking & front coding	7.9 5.9
	collection (text, xml markup etc)	3600.0
	collection (text)	960.0
	term incidence matrix	40,000.0
	postings, uncompressed (32-bit words)	400.0
	postings, uncompressed (20 bits)	250.0
	postings, variable byte encoded	116.0
	postings, γ encoded	101.0

The vector space model for scoring

How do we quantify the similarity between two documents in this vector space? A first attempt might consider the magnitude of the vector difference between two document vectors. This measure suffers from a drawback: two documents with very similar content can have a significant vector difference simply because one is much longer than the other. Thus the relative distributions of terms may be identical in the two documents, but the absolute term frequencies of one may be far larger.



► **Figure 6.4** Cosine similarity illustrated. $\text{sim}(d_1, d_2) = \cos\theta$.



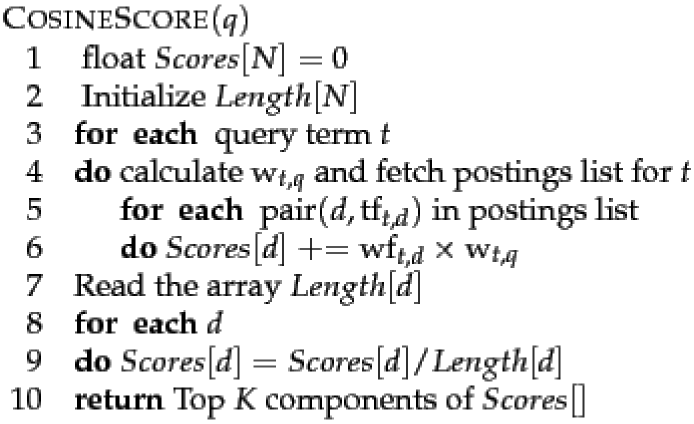
term	SaS	PaP	WH
affection	0.996	0.993	0.847
jealous	0.087	0.120	0.466
gossip	0.017	0	0.254

▶ **Figure 6.6** Term vectors for the three novels of Figure 6.12. These are based on raw term frequency only and are normalized as if these were the only terms in the collection. (Since affection and jealous occur in all three documents, their tf-idf weight would be 0 in most formulations.)

Now consider the cosine similarities between pairs of the resulting three-dimensional vectors. A simple computation shows that sim($\vec{v}$(SAS), $\vec{v}$(PAP)) is 0.999, whereas sim($\vec{v}$(SAS), $\vec{v}$(WH)) is 0.888; thus, the two books authored by Austen (SaS and PaP) are considerably closer to each other than to Brontë's *Wuthering Heights*. In fact, the similarity between the first two is almost perfect (when restricted to the three terms we consider). Here we have considered

Computing vector scores

In a typical setting we have a collection of documents each represented by a vector, a *free text query* represented by a vector, and a positive integer K. We seek the K documents of the collection with the highest vector space scores on the given query. We now initiate the study of determining the K documents with the highest vector space scores for a query. Typically, we seek these K top documents in ordered by decreasing score; for instance many search engines use $K = 10$ to retrieve and rank-order the first page of the ten best results. Here we give the basic algorithm for this computation; we develop a fuller treatment of efficient techniques and approximations in Chapter 7 .

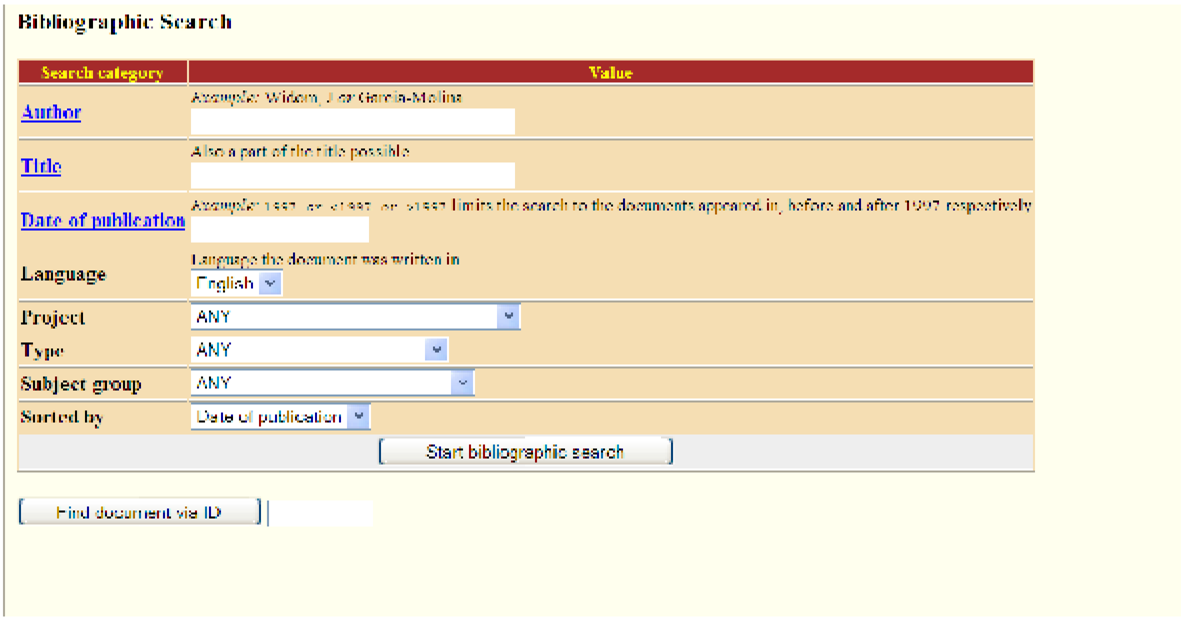


```
CosineScore($q$)
float $Scores[N] = 0$
Initialize $Length[N]$
for each query term $t$
do calculate $w_{t,q}$ and fetch postings list for $t$
    for each pair($d$, $tf_{t,d}$) in postings list
    do $Scores[d] += wf_{t,d} \times w_{t,q}$
Read the array $Length[d]$
for each $d$
do $Scores[d] = Scores[d] / Length[d]$
return Top $K$ components of $Scores[]$
```

Figure 6.14: The basic algorithm for computing vector space scores.

Parametric and zone indexes

This metadata would generally include *fields* such as the date of creation and the format of the document, as well the author and possibly the title of the document. The possible values of a field should be thought of as finite - for instance, the set of all dates of authorship.

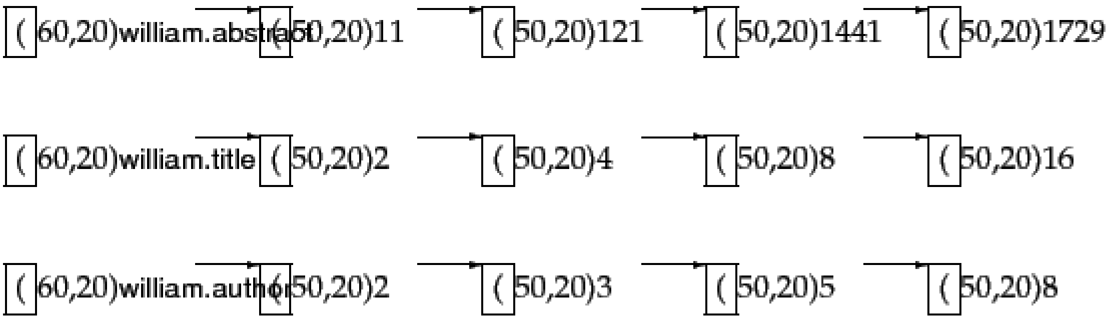


Bibliographic Search

Search category	Value
Author	Example: Widom, J or Garcia-Molina
Title	Also a part of the title possible
Date of publication	Example: 1997 or <1997 or >1997 limits the search to the documents appeared in, before and after 1997 respectively
Language	Language the document was written in English
Project	ANY
Type	ANY
Subject group	ANY
Sorted by	Date of publication

Parametric search.In this example we have a collection with fields allowing us to select publications by zones such as Author and fields such as Language.

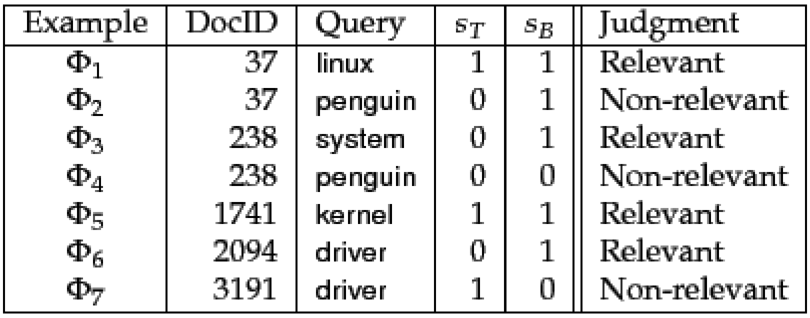
We may build a separate inverted index for each zone of a document, to support queries such as ``find documents with merchant in the title and william in the author list and the phrase gentle rain in the body''. This has the effect of building an index that looks like Figure 6.2. Whereas the dictionary for a parametric index comes from a fixed vocabulary (the set of languages, or the set of dates), the dictionary for a zone index must structure whatever vocabulary stems from the text of that zone.



▶ **Figure 6.1** Basic zone index ; zones are encoded as extensions of dictionary entries.

Learning weights

1. We are provided with a set of *training examples*, each of which is a tuple consisting of a query q and a document d, together with a relevance judgment for d on q. In the simplest form, each relevance judgments is either *Relevant* or *Non-relevant*. More sophisticated implementations of the methodology make use of more nuanced judgments.
2. The weights g_i are then ``learned'' from these examples, in order that the learned scores approximate the relevance judgments in the training examples.



Example	DocID	Query	s_T	s_B	Judgment
Φ_1	37	linux	1	1	Relevant
Φ_2	37	penguin	0	1	Non-relevant
Φ_3	238	system	0	1	Relevant
Φ_4	238	penguin	0	0	Non-relevant
Φ_5	1741	kernel	1	1	Relevant
Φ_6	2094	driver	0	1	Relevant
Φ_7	3191	driver	1	0	Non-relevant

Figure 6.5: An illustration of training examples.

For each training example Φ_j we have Boolean values $s_T(d_j, q_j)$ and $s_B(d_j, q_j)$ that we use to compute a score from (14)

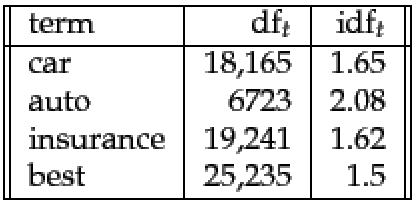
$$score(d_j, q_j) = g \cdot s_T(d_j, q_j) + (1 - g) \cdot s_B(d_j, q_j). \quad (15)$$

Term frequency and weighting

Inverse document frequency

Raw term frequency as above suffers from a critical problem: all terms are considered equally important when it comes to assessing relevancy on a query. In fact certain terms have little or no discriminating power in determining relevance.

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low. Figure 6.8 gives an example of idf's in the Reuters collection of 806,791 documents; in this example logarithms are to the base 10. In fact, as we will see in Exercise 6.2.2 , the precise base of the logarithm is not material to ranking. We will give on page 11.3.3 a justification of the particular form in Equation 21.



term	df_t	idf_t
car	18,165	1.65
auto	6723	2.08
insurance	19,241	1.62
best	25,235	1.5

► **Figure 6.3** Example of idf values. Here we give the idf's of terms with various frequencies in the Reuters collection of 806,791 documents.

Tf-idf weighting

We now combine the definitions of term frequency and inverse document frequency, to produce a composite weight for each term in each document. The *tf-idf* weighting scheme assigns to term t a weight in document d given by

$$\text{tf-idf}_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t. \tag{22}$$

In other words, $\text{tf-idf}_{t,d}$ assigns to term t a weight in document d that is

1. highest when t occurs many times within a small number of documents (thus lending high discriminating power to those documents);

2. lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal);
3. lowest when the term occurs in virtually all documents.

At this point, we may view each document as a *vector* with one component corresponding to each term in the dictionary, together with a weight for each component that is given by (22). For dictionary terms that do not occur in a document, this weight is zero. This vector form will prove to be crucial to scoring and ranking; we will develop these ideas in Section 6.3 . As a first step, we introduce the *overlap score measure*: the score of a document d is the sum, over all query terms, of the number of times each of the query terms occurs in d. We can refine this idea so that we add up not the number of occurrences of each query term t in d, but instead the tf-idf weight of each term in d.

$$\mbox{Score}(q,d) = \sum_{t\in q} \mbox{tf-idf}_{t,d}. \tag{23}$$

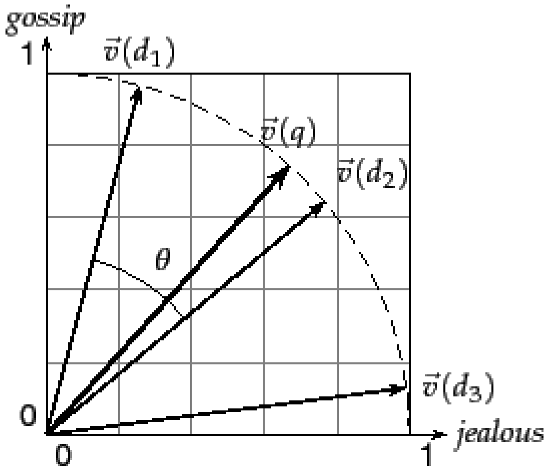
In Section 6.3 we will develop a more rigorous form of Equation 23.

The vector space model for scoring

Dot products

We denote by $\vec{V}(d)$ the vector derived from document d, with one component in the vector for each dictionary term. Unless otherwise specified, the reader may assume that the components are computed using the tf-idf weighting scheme, although the particular weighting scheme is immaterial to the discussion that follows. The set of documents in a collection then may be viewed as a set of vectors in a vector space, in which there is one axis for each term. This representation loses the relative ordering of the terms in each document; recall our example from Section 6.2 (page □), where we pointed out that the documents Mary is quicker than John and John is quicker than Mary are identical in such a *bag of words* representation.

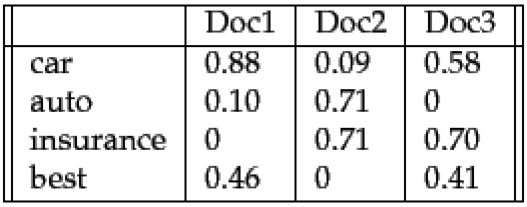
How do we quantify the similarity between two documents in this vector space? A first attempt might consider the magnitude of the vector difference between two document vectors. This measure suffers from a drawback: two documents with very similar content can have a significant vector difference simply because one is much longer than the other. Thus the relative distributions of terms may be identical in the two documents, but the absolute term frequencies of one may be far larger.



► **Figure 6.4** Cosine similarity illustrated. $\text{sim}(d_1, d_2) = \cos\theta$.

To compensate for the effect of document length, the standard way of quantifying the similarity between two documents d_1 and d_2 is to compute the *cosine similarity* of their vector representations $\vec{V}(d_1)$ and $\vec{V}(d_2)$

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|}, \tag{24}$$



	Doc1	Doc2	Doc3
car	0.88	0.09	0.58
auto	0.10	0.71	0
insurance	0	0.71	0.70
best	0.46	0	0.41

Figure 6.11: Euclidean normalized tf values for documents in Figure 6.9 .

Variant tf-idf functions

Sublinear tf scaling

It seems unlikely that twenty occurrences of a term in a document truly carry twenty times the significance of a single occurrence. Accordingly, there has been considerable research into variants of term frequency that go beyond counting the number of occurrences of a term. A common modification is to use instead the logarithm of the term frequency, which assigns a weight given by

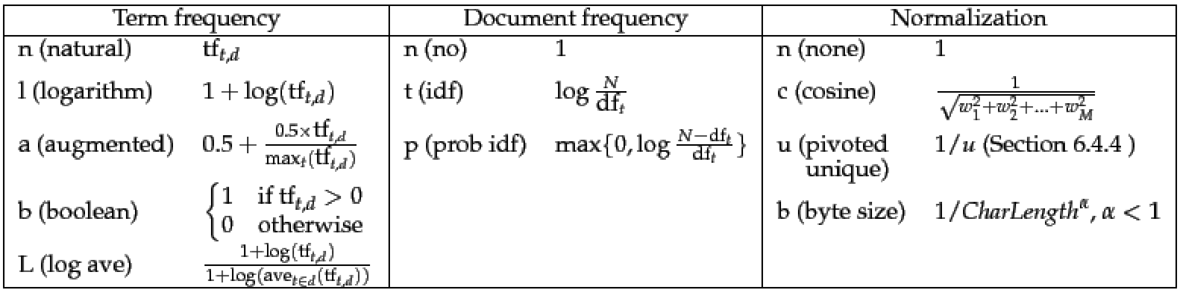
$$\mathrm{wf}_{t,d} = \begin{cases} 1 + \log \mathrm{tf}_{t,d} & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{28}$$

In this form, we may replace tf by some other function wf as in (28), to obtain:

$$\text{wf-idf}_{t,d} = \mathrm{wf}_{t,d} \times \mathrm{idf}_t. \tag{29}$$

Document and query weighting schemes

Equation 27 is fundamental to information retrieval systems that use any form of vector space scoring. Variations from one vector space scoring method to another hinge on the specific choices of weights in the vectors $\vec{V}(d)$ and $\vec{V}(q)$.



Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5 + \frac{0.5\times\mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$ (Section 6.4.4)
b (boolean)	$\begin{cases} 1 & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/\mathit{CharLength}^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

► **Figure 6.7** SMART notation for tf-idf variants. Here *CharLength* is the number of characters in the document.

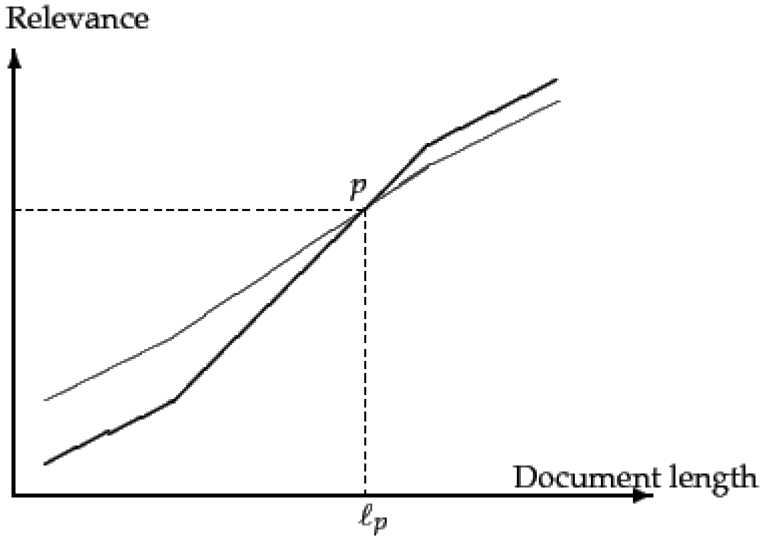


Figure 6.16: Pivoted document length normalization.

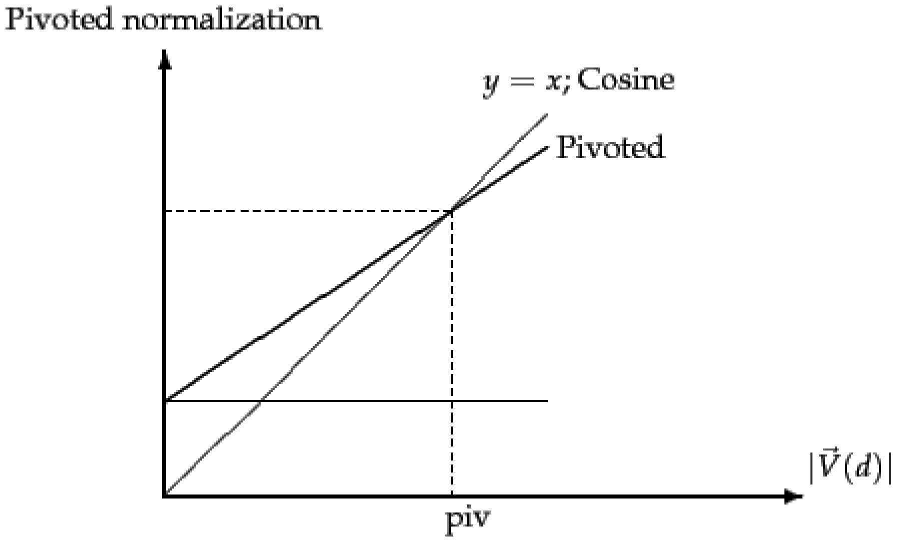


Figure 6.17: Implementing pivoted document length normalization by linear scaling.

Of course, pivoted document length normalization is not appropriate for all applications. For instance, in a collection of answers to frequently asked questions (say, at a customer service website), relevance may have little to do with document length.