Scoring, Term Weighting and the Vector Space Model

Chien Chin Chen

Department of Information Management
National Taiwan University

Indexing and Boolean Retrieval Model (1/8)

- Now … we know how to extract terms from documents.
- The next step of building an information retrieval system is to **index** the documents that each term occurs.
- Before indexing … we usually assign each document a unique serial number, known as the ***document identifier*** (`docID`).
- Then … the simplest method of indexing is to construct a ***binary term-document incidence matrix***.

Indexing and Boolean Retrieval Model (2/8)

term	docID	1	2	3	4	5	6
	antony	1	1	0	0	0	1
	brutus	1	1	0	1	0	0
	caesar	1	1	0	1	1	1
	calpurnia	0	1	0	0	0	0
	cleopatra	1	0	0	0	0	0
	mercy	1	0	1	1	1	1
	worser	1	0	1	1	1	0

1 if a document contains a word

- Using this matrix, an information retrieval system can easily answer user's **Boolean queries**.
  - For example: to answer the query "*Brutus* AND *Caesar* AND NOT *Calpurnia*".

 110100
AND 110111
AND 101111
 = 100100

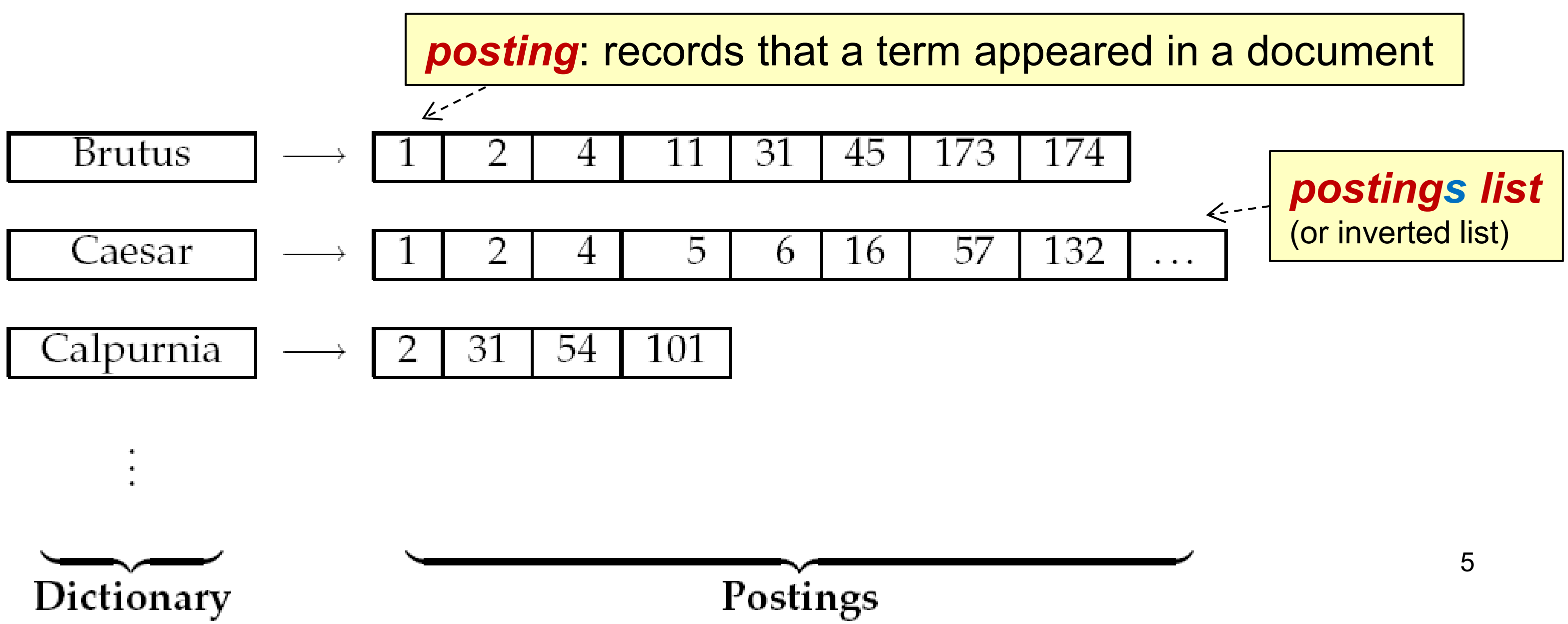
query terms are combined with the operators **AND**, **OR**, and **NOT**.

Indexing and Boolean Retrieval Model (3/8)

- Is term-document-matrix based indexing feasible??
  - **No!!**
  - Suppose that we have 1 million (1,000,000) documents for indexing.
  - And the collection contains 500,000 distinct terms.
  - Then … the matrix will have 500,000 x 1,000,000 = 5×10^{11} entries.
  - If 1 bit per entry, **the matrix will cost around 58GB memory**!!
  - Zipf's law tells us that **the matrix will be very sparse**.
    - It has few non-zero entries.

Indexing and Boolean Retrieval Model (4/8)

- A better way of indexing is to record only the things that do occur – ***inverted index***.
  - Sometimes referred as ***inverted file***.
  - Consists of two parts: ***dictionary*** and ***postings***.



Indexing and E (5/8)

- The input to (inverted) index construction is a list of normalized tokens for each document.
- Then, we **sort** this list so that the terms are alphabetical.
- Next, multiple occurrences of the same term from the same document are merged.
- Instances of the same term are then grouped.
  - The result is split into a dictionary and postings.

term	docID
I	1
did	1
enact	1
julius	1
casear	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

⟹

	term	docID
	ambitious	2
	be	2
	brutus	1
	brutus	2
	capitol	1
	caesar	1
	caesar	2
✖	caesar	2
	did	1
	enact	1
	hath	1
	I	1
✖	I	1
	i'	1
	it	2
	julius	1
	killed	1
✖	killed	1
	let	2
	me	1
	noble	2
	so	2
	the	1
	the	2
	told	2
	you	2
	was	1
	was	2
	with	2

⟹

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Indexing and Boolean Retrieval Model (6/8)

- The dictionary also records some statistics, such as the number of documents which contain each term (document frequency).
  - Which can be used to rank retrieval documents.

- Postings are much larger than dictionary.
  - So … in general, we keep the dictionary in memory.
  - And posting lists are normally kept on disk.

Indexing and Boolean Retrieval Model (7/8)

- How to process Boolean queries using an inverted index.
- Consider processing the simple *conjunctive* query:

  Brutus AND *Calpurnia*

1. Locate *Brutus* in the dictionary.
2. Retrieve its postings.
3. Locate *Calpurnia* in the dictionary.
4. Retrieve its postings.
5. **Intersect** (merge) the two postings lists.

Indexing and Boolean Retrieval Model (8/8)

- The intersection operation needs to be efficient.
- Here we present an effective merge algorithm that requires the postings being sorted by docID.

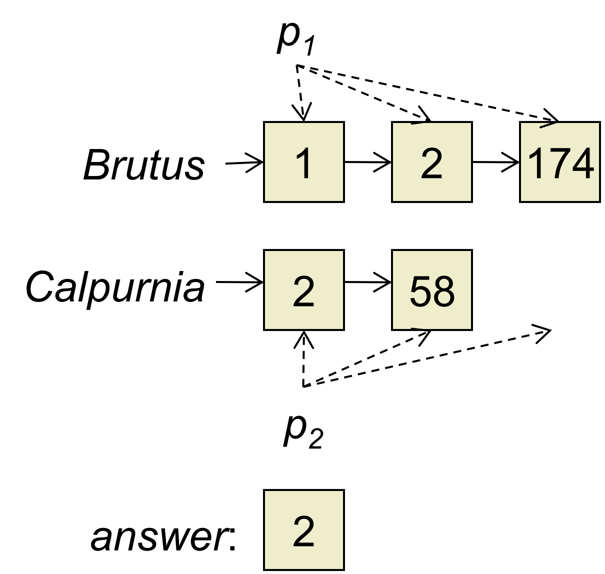
```
INTERSECT(p1,p2)
  answer ← <>
  while p1 ≠ NULL and p2 ≠ NULL

    if docID(p1) == docID(p2)
      ADD(answer,docID(p1))
      p1 ← next(p1)
      p2 ← next(p2)

    else if docID(p1) < docID(p2)
      p1 ← next(p1)

    else
      p2 ← next(p2)

  return answer
```



the intersection takes $O(x+y)$, where x and y are the lengths of the postings lists, respectively

The Problem of Boolean Retrieval Model

- Document either matches or does not match a Boolean query.
- In the case of large document collections, the resulting numbers matching can be very large.
- Accordingly, it is essential for information retrieval systems to **rank-order** the documents matching a query.
  - According to a certain matching score.
- Here, we present two ways of assigning a score to a query-document pair:
  - Parametric and zone indexes.
  - Term weighting and vector space model.

Parametric and Zone Indexes (1/2)

- In practice, most documents have additional structure and ***metadata***.
  - Metadata – specific forms of data about data (a document).
    - Such as *language*, *authors*, *title*, *keyword*, and *data of publication*.

- Generally, there are two types of metadata:
  - ***Field*** – the possible values of a field should be **finite**.
    - Such as the format of the document, date of publication.
    - There is one ***parametric index*** for each field.
  - ***Zone*** – the contents of a zone can be **arbitrary (unbounded amount) free text**.
    - Such as document titles.
    - We can build a separate inverted index for each zone – ***zone index***.

Parametric and Zone Indexes (2/2)

- Users can then submit specific queries to retrieve documents effectively.
  - "Find documents authored by William Shakespeare in 1601, containing the term Yorick".

zone: author

field: date

general (zone) index

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

Start bibliographic search

Weighted Zone Scoring (1/2)

- Weighted zone scoring assign a score in [0, 1] to a pair (q, d) by computing a linear combination of zone scores.

$$score = \sum_{i=1}^{l} g_i s_i$$

each document has l zones

  - s_i is the Boolean score denoting a match (or absence) between q and the ith zone.
  - g_i in [0,1] such that $\Sigma^{l}_{i=1} g_i = 1$.

- With the score, we can rank documents.
  - Therefore, this method is sometimes referred to also as ***ranked Boolean retrieval***.

Weighted Zone Scoring (2/2)

- Example:
  - Each document in a collection has three zones: *author*, *title*, and *body*.
  - Weighted zone scoring would require three weights g_1, g_2, and g_3, respectively corresponding to the *author*, *title*, and *body* zones.
    - $g_1 = 0.2$, $g_2 = 0.3$, and $g_3 = 0.5$.
    - The *body* contributes as much as either *author* and *title*.

- How do we determine the weights g_i for weighted zone scoring?
  - By an expert;
  - Learned using training examples.

Term Frequency and Weighting (1/2)

- Weighted zone scoring hinges on whether or not a query term(**s**) is present in a zone within a document.
  - Entire presence … or … absence … unreasonable.

- A more logical consideration:
  - A document that mentions a query term(**s**) **more** often has more to do with that query and therefore should receive a higher score.
  - A ***scoring mechanism*** then is to compute a score of the **matches** between query terms and the document.
    - A match score → the weight of the matched term in documents.

- ***Free text query***:
  - The terms of the query are typed freeform into the search interface, without any connecting search operators (such as Boolean operators).
  - Very popular on the Web.

Term Frequency and Weighting (2/2)

- ***Term Frequency*** (***TF***):
  - The weight of a term depends on the number of occurrences of the term in the document.
  - Notation: $tf_{t,d}$ — the number of occurrences of term *t* in document *d*.

can be term frequency or determined by other weighting schemes

- The ***bag of words*** model:
  - A common representation of documents.
  - The representation of a document *d* is the **set** of weights of its terms.
    - Example: *"term i and term j are synonyms"* → { <term,2>, <and, 1>, …}
    - Set: **the ordering of the terms is ignored**!!
    - *"Mary is quicker than John"* == *"John is quicker than Mary"*

Inverse Document Frequency (1/4)

- A critical problem of term frequency weighting scheme:
  - Each term occurrence is considered **equally important**.
  - *“term i and term j are synonyms”*

weight contribution: 1 1 1 1 1 1 1

- In fact, certain terms have little or **no discriminating power**.
  - For instance, a collection of documents on the auto industry is likely to have the term *‘auto’* in almost every document.
  - We need a mechanism for reducing the effect of terms that **occur too often** in the collection.

Inverse Document Frequency (2/4)

- ***Document frequency***:
  - Notation: df_t
  - The **number of documents** in the collection that contain a term *t*.

- ***Inverse document frequency*** **(*IDF*)**:
  - Notation:
    $$idf_t = \log \frac{N}{df_t}$$
    the number of documents in a collection
  - The *idf* of a rare term is high, and is likely to be low for a frequent term.

Inverse Document Frequency (3/4)

- An alternative to document frequency — ***collection frequency*** (***CF***).
  - The total number of occurrences of a term in the collection.
  - But … the purpose of term scoring is to discriminate **between documents**.
  - It is better to use a document-level statistic (DF) than to use a collection-wide statistic for term weighting.

can be a general term appearing in many documents

can be a discriminating term appearing in a certain of documents

Word	CF	DF
'try'	10422	8760
'insurance'	10440	3997

Inverse Document Frequency (4/4)

- Example of IDF values of terms in the Reuters collection of 806,791 documents.

term	DF	IDF
‘car’	18,165	1.65
‘auto’	6,723	2.08
‘insurance’	19,241	1.62
‘best’	25,235	1.5

TF-IDF weighting

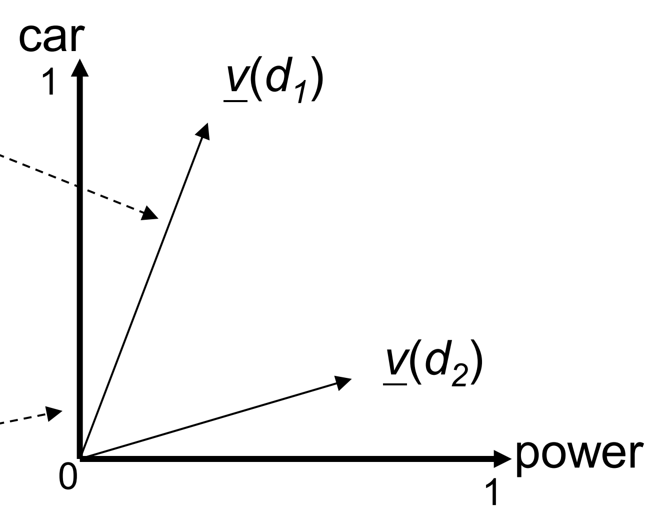
- ***TF-IDF*** combines the concept of term frequency and inverse document frequency to assign the weight of term *t* in document *d* as follows:
  - $tf\text{-}idf_{t,d} = tf_{t,d} \times idf_t$.
  - The weight of term *t* in document *d* is:
    - **High**, when *t* occurs many times in *d* and appears within a small number of documents.
    - **Low**, when *t* is a rare term in *d* and occurs in virtually all documents in the collection.
- A simple scoring mechanism of a query *q* to a document *d* – the ***overlap score measure***:
  - $\text{score}(q,d) = \sum_{t \text{ in } q} tf\text{-}idf_{t,d}$

Vector Space Model (1/6)

- We may view each document (or query) as a vector:
  - One component (dimension) corresponding to each term in the dictionary.
    - Even with stemming, we may have 20,000+ dimensions!!
  - With a weight for each component that is given by a weighting scheme (such as tf-idf).
    - The weight of a term determines its importance in a document.

The set of documents in a collection can be viewed as a set of vectors in a vector space.

Each axis for the vector space represents a term in the dictionary.



Vector Space Model (2/6)

- This representation *loses* the *relative ordering* of the terms in each document.
  - Is identical to the *bag of words* representation.

- How do we quantify the similarity between two documents in this vector space?
  - The **component difference** between the vectors.

$$\sum_{t \in dictionary} | (\underline{V}(d_1)_t - \underline{V}(d_2)_t) |$$

weight of term *t* in document d_2

    - Content-similar documents may have a significant vector difference due to the **different document length**.

  - A better and popular method is …

Vector Space Model (3/6)

- ***Cosine similarity***:

$$sim(d_1, d_2) = \frac{\underline{V}(d_1) \cdot \underline{V}(d_2)}{|\underline{V}(d_1)||\underline{V}(d_2)|}$$

vector length

$$\sqrt{\sum_{t \in dictionary} \underline{V}(d_1)_t * \underline{V}(d_1)_t}$$

inner product of vectors

$$\sum_{t \in dictionary} \underline{V}(d_1)_t * \underline{V}(d_2)_t$$

- The effect of the denominator is to normalized vectors to **unit vector**.
  - To compensate for the effect of document length.
- The range of cosine similarity is on [0,1].
  - 1 → identical.
  - 0 → orthogonal.
- This measure is the cosine of the angle θ between the two vectors.

Vector Space Model (4/6)

- The cosine similarity can be rewritten as

$$sim(d_1, d_2) = \frac{\underline{V}(d_1) \cdot \underline{V}(d_2)}{|\underline{V}(d_1)||\underline{V}(d_2)|}$$

$$= \frac{\underline{V}(d_1)}{|\underline{V}(d_1)|} \cdot \frac{\underline{V}(d_2)}{|\underline{V}(d_2)|}$$

- That is **the <u>inner product</u> of the <u>unit vectors</u>**.

Vector Space Model (5/6)

- What use is the similarity measure between documents?
  - **To find similar documents**, a popular function of many search engines.
    - For a user specified document *d*, we compute the cosine similarities between $\underline{V}(d)$ and each of $\underline{V}(d_1)$, …, $\underline{V}(d_N)$.
    - Then picking off the highest resulting similarity value documents.
  - **To cluster documents** into content coherent clusters.

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

term-document matrix

term vector of a document

So high, due to the same author

sim(SaS, PaP) = 0.999
sim(SaS, WH) = 0.888

Vector Space Model (6/6)

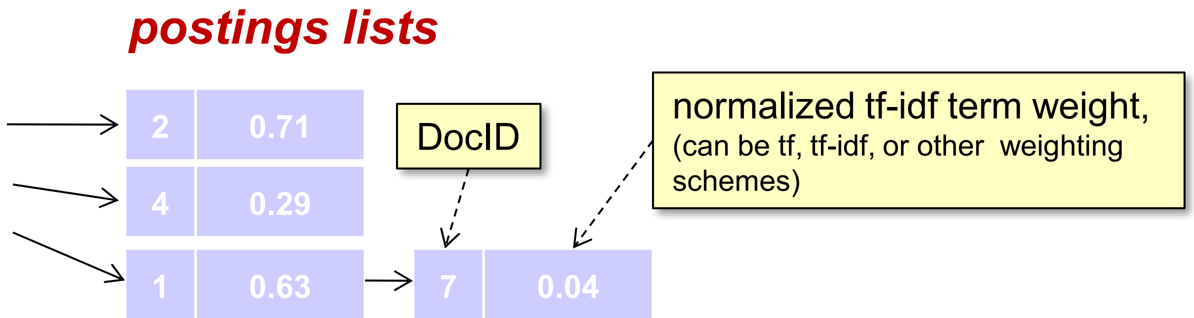
- Queries as vectors.
  - By viewing a query as a “bag of words”, we can treat it as **a very short document**.
  - Consequentially, we can use the **cosine similarity** between the query vector and a document vector as a measure of the score of the document for that query.
    - score = sim(d,q).
  - The scores can then be used to **rank** and select top-scoring documents for a query.
  - A document may have a high score for a query **even if it does not contain all query terms!!**
    - Contrast to Boolean search.

Computing Vector Scores (1/2)

- Here we show the basic algorithm for query-document score calculation.

dictionary

Term	Doc. Freq.
ambitious	1
be	1
brutus	2
…	…



CosineScore(q)
  float $Score[N]$ = 0
  calculate normalized tf-idf weight for each query term
  for each query term t
    fetch postings list for t
    for each pair (d, $w_{t,d}$) in postings list
      add $w_{t,d}$ x $w_{t,q}$ to $Scores[d]$

  return Top K documents of $Scores[]$

Computing Vector Scores (2/2)

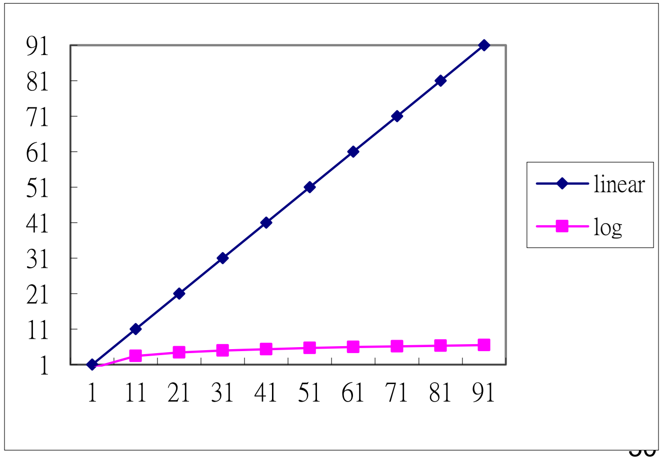
- It is wasteful to store tf-idf weights in the postings lists.
  - Each posting entry requires a floating point number.
- Moreover, the number of documents in an information retrieval system can grow.
  - The pre-calculated tf-idf weights may not reflect the latest idf information.
- Some systems only store term frequency for each postings entry.
  - Each entry then only requires an integer.
- This methodology can save space dramatically, but …
  - Need to calculate term weights of documents online.
  - To normalized weights, not only query terms but also other terms need for weight calculations.

Variants in TF-IDF Functions (1/3)

- Twenty occurrences of a term in a document truly carry twenty times the significance of a single occurrence??
  - We observe higher term frequencies in documents, merely because longer documents tend to repeat the same words over and over again.

- **Sub-linear TF scaling**:
  - A common modification of TF is to use the logarithm of the term frequency.

$$wf_{t,d} = \begin{cases} 1+\log tf_{t,d} & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$$



  - Then, replace TF-IDF as WF-IDF:
    - $wf\text{-}idf_{t,d} = wf_{t,d} * idf_t$.

Variants in TF-IDF Functions (2/3)

- **Maximum TF normalization**:
  - To normalize the TF weights of all terms by the maximum TF in that document.
  - Let $tf_{\max}(d) = \max_{\tau \in d} tf_{\tau,d}$,
  - Then, $ntf_{t,d} = a + (1-a)\dfrac{tf_{t,d}}{tf_{\max}(d)}$

To damp the contribution of tf in term weight

a is in [0, 1] and is generally set to 0.4, or 0.5 suggested by Gerard Salton

Variants in TF-IDF Functions (3/3)

- SMART notation to document and query weighting schemes:
  - ddd.qqq

    Term weighting scheme for document,
    the **first** d specifies the term frequency,
    the **second** specifies the document frequency,
    the **third** is the form of normalization.

  - Example: lnc.ltc.

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