Introduction to

Information Retrieval

Lecture 5: Scoring, Term Weighting and the Vector Space Model

Agenda

- Ranked retrieval
- Scoring documents
- Term frequency
- Collection statistics
- Weighting schemes
- Vector space scoring

Ranked retrieval

- Thus far, our queries have all been Boolean.
  - Documents either match or don’t.
- Good for expert users with precise understanding of their needs and the collection
- Not good for the majority of users.
  - Most users incapable of writing Boolean queries (or they are, but they think it’s too much work).
  - Most users don’t want to wade through 1000s of results.
    - This is particularly true of web search.

Problem with Boolean search: feast or famine

- Boolean queries often result in either too few (=0) or too many (1000s) results.
- Query 1: "*standard user dlink 650*" → 200,000 hits
- Query 2: "*standard user dlink 650 no card found*": 0 hits
- It takes a lot of skill to come up with a query that produces a manageable number of hits.
  - AND gives too few; OR gives too many

Ranked retrieval models

- Rather than a set of documents satisfying a query expression, in ranked retrieval, the system returns an ordering over the (top) documents in the collection for a query
- Free text queries: Rather than a query language of operators and expressions, the user's query is just one or more words in a human language
- *Ranked retrieval has been associated with free text queries.*

Feast or famine: not a problem in ranked retrieval

- When a system produces a ranked result set, large result sets are not an issue
  - Indeed, the size of the result set is not an issue
  - We just show the top k (≈ 10) results
  - We don't overwhelm the user

  - ***Premise: the ranking algorithm works***

Scoring as the basis of ranked retrieval

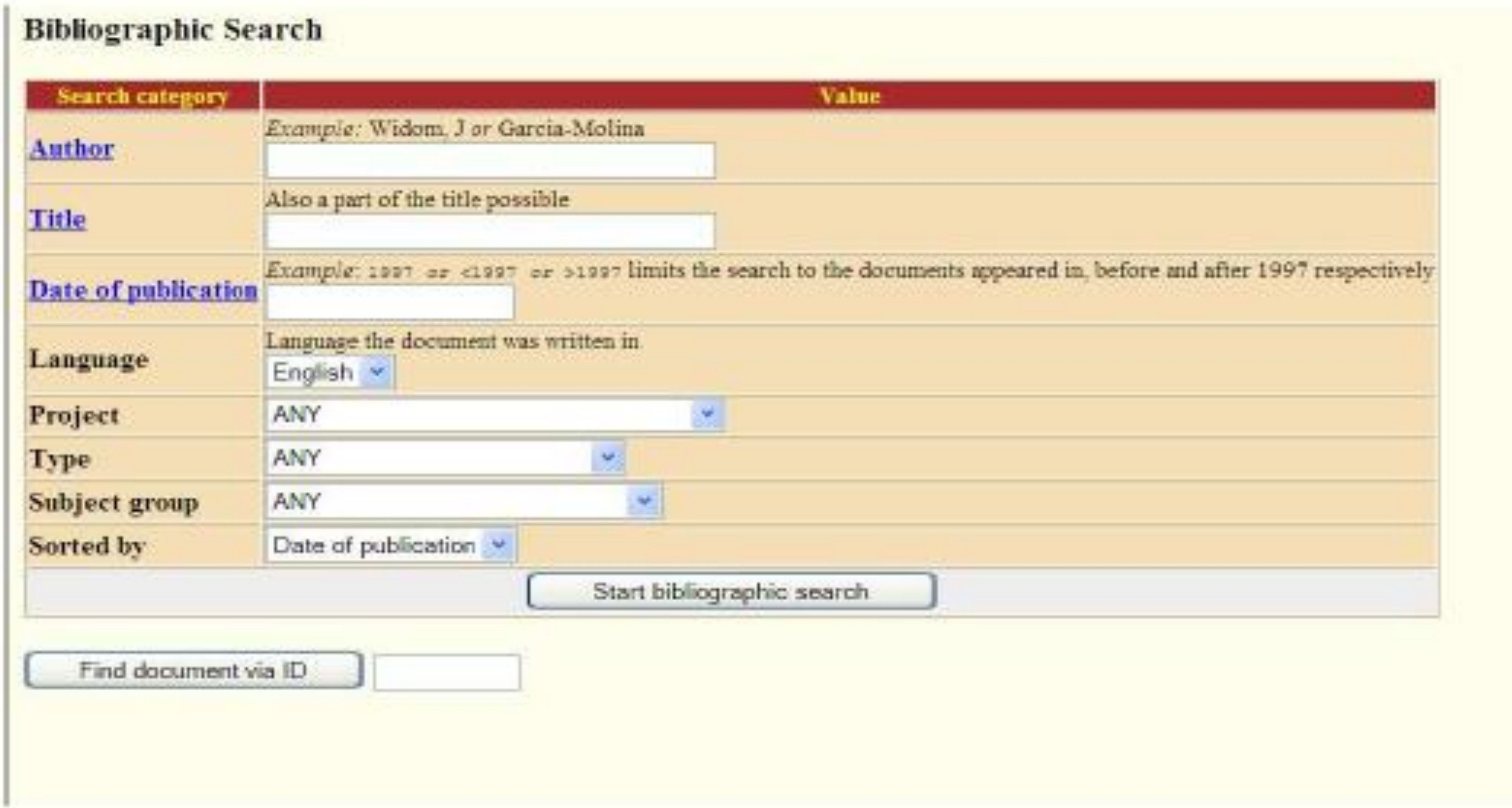
- We wish to return in order the documents most likely to be useful to the searcher
- *How can we rank-order the documents in the collection with respect to a query?*
- Assign a score – say in [0, 1] – to each document
- This score measures how well document and query "match".

Parametric and Zone indexes

Metadata, Fields, Zones

- Documents can have metadata and fields
  - E.g., title of document, author of document, date of creation
- Zones similar to fields, but can contain arbitrary text
  - E.g., abstract, introduction, … of a research paper

- We can have an index for each field/zone
  - To support queries like “documents having *merchant* in the title and *william* in the author list”
  - Either separate index for each field/zone, or part of the same index



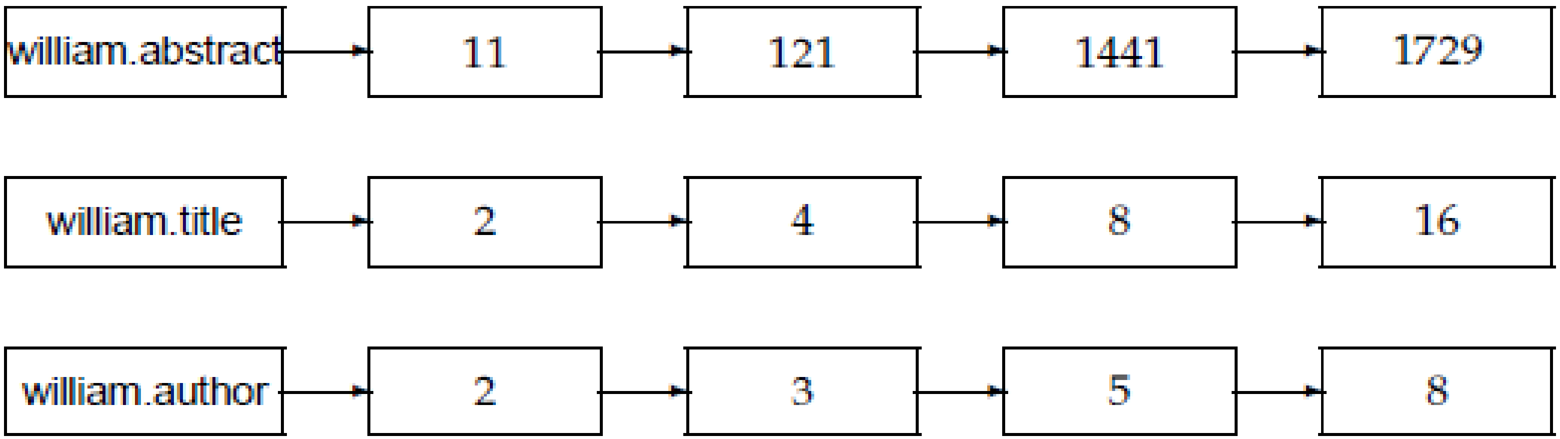
Bibliographic Search

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

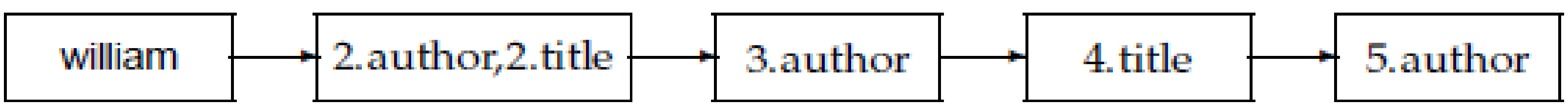
Start bibliographic search

Find document via ID

► **Figure 6.1** 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.



► **Figure 6.2** Basic zone index ; zones are encoded as extensions of dictionary entries.



► **Figure 6.3** Zone index in which the zone is encoded in the postings rather than the dictionary.

Weighted zone scoring

- Given a Boolean query q and a document d
  - Compute a 'zone match score' in [0,1] for each zone/field of d with q
  - Compute linear combination of zone match scores, where each zone assigned a weight (sum of weights equal to 1.0)
  - Sometimes called 'ranked Boolean retrieval'
- How to decide the weights?
  - Option 1: Specified by experts, e.g., match in "title" has higher significance than match in "body"
  - Option 2: Learn from training examples – application of Machine Learning

- Given a Boolean query *q* and a document *d*, weighted zone scoring assigns to the pair (*q*, *d*) a score in the interval [0, 1], by computing a linear combination of *zone scores*, where each zone of the document contributes a Boolean value. More specifically, consider a set of documents each of which has ℓ zones. Let $g_1, \ldots, g_\ell \in [0, 1]$ such that $\sum_{i=1}^{\ell} g_i = 1$. For $1 \leq i \leq \ell$,

  For $1 \leq i \leq \ell$, let s_i be the Boolean score denoting a match (or absence

  thereof) between *q* and the *i*th zone.
- For instance, the Boolean score from a zone could be 1 if all the query term(s) occur in that zone, and zero otherwise.
- Then, the weighted zone score is defined to be

$$\sum_{i=1}^{\ell} g_i s_i.$$

- Consider the query Shakespeare in a collection in which each document has three zones: *author, title* and *body*.
- The Boolean score function for a zone takes on the value 1 if the query term Shakespeare is present in the zone, and zero otherwise.
- Weighted zone scoring in such a collection would require three weights $g1$, $g2$ and $g3$, respectively corresponding to the *author, title* and *body* zones.
- Suppose we set $g1 = 0.2$, $g2 = 0.3$ and $g3 = 0.5$ (so that the three weights add up to 1);
- Thus if the term Shakespeare were to appear in the *title* and *body* zones but not the *author* zone of a document, the score of this document would be 0.8.

- **Example 6.1:** Consider the query shakespeare in a collection in which each document has three zones: *author, title* and *body*. The Boolean score function for a zone takes on the value 1 if the query term shakespeare is present in the zone, and zero otherwise. Weighted zone scoring in such a collection would require three weights $g1$, $g2$ and $g3$, respectively corresponding to the *author, title* and *body* zones.

- **In the example above with weights $g1 = 0.2$, $g2 = 0.31$ and $g3 = 0.49$, what are all the distinct score values a document may get?**

- Weighted zone scores can be directly computed from inverted indexes.
- The algorithm treats the case when the query *q* is a two term query consisting of query terms *q*1 and *q*2, and the Boolean function is AND: 1 if both query terms are present in a zone and 0 otherwise.

Implement the computation of weighted zone scores

```
INTERSECT(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
       then ADD(answer, docID(p1))
            p1 ← next(p1)
            p2 ← next(p2)
       else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
  return answer
```

▶ Figure 1.6 Algorithm for the intersection of two postings lists p_1 and p_2.

Algorithm for computing the weighted zone score from two postings lists.

```
ZONESCORE(q1, q2)
  float scores[N] = [0]
  constant g[ℓ]
  p1 ← postings(q1)
  p2 ← postings(q2)
  // scores[] is an array with a score entry for each document, initialized to zero.
  // p1 and p2 are initialized to point to the beginning of their respective postings.
  // Assume g[] is initialized to the respective zone weights.
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
       then scores[docID(p1)] ← WEIGHTEDZONE(p1, p2, g)
            p1 ← next(p1)
            p2 ← next(p2)
       else if docID(p1) < docID(p2)
              then p1 ← next(p1)
              else p2 ← next(p2)
  return scores
```

Learning weights

- How do we determine the weights *gi* for weighted zone scoring?
  - These weights could be specified by an expert (or, in principle, the user);
  - these weights are “learned” using training examples
- *Machine-learned relevance*.

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*.
2. The weights *gi* are then “learned” from these examples, in order that the learned scores approximate the relevance judgments in the training examples.

- We now consider a simple case of weighted zone scoring, where each document has a *title* zone and a *body* zone.

$$score(d,q) = g \cdot s_T(d,q) + (1-g)s_B(d,q).$$

- T*raining set examples* - each of which is a triple of the form

$$\Phi j = (dj, qj, r(dj, qj)).$$

- Human editor delivers a relevance judgment $r(dj, qj)$ that is either *Relevant* or *Non-relevant*.

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.

s_T	s_B	Score
0	0	0
0	1	$1-g$
1	0	g
1	1	1

► **Figure 6.6** The four possible combinations of s_T and s_B.

- For each training example Φj we have Boolean values *sT*(*dj*, *qj*) and *sB*(*dj*, *qj*) that we use to compute a score

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

- We now compare this computed score to the human relevance judgment for the same document-query pair (*dj*, *qj*)
- Suppose that we define the error of the scoring function with weight *g* as

$$\varepsilon(g, \Phi_j) = (r(d_j, q_j) - score(d_j, q_j))^2,$$

- where we have quantized the editorial relevance judgment r(dj, qj) to 0 or 1.
- Then, the total error of a set of training examples is given by

$$\sum_j \varepsilon(g, \Phi_j).$$

- The problem of learning the constant *g* from the given training examples then reduces to picking the value of *g* that minimizes the total error

Learning weights (simple machine learning)

Assume only two zones *title* (T) and *body* (B) with zone weights g and $1 - g$, respectively.

Example	DocID d	Query	s_T	s_B	Judgment (human expert)	r (quantized judgment)
ϕ_1	37	linux	1	1	Relevant	1
ϕ_2	37	penguin	0	1	Non-relevant	0
ϕ_3	238	system	0	1	Relevant	1
ϕ_4	238	penguin	0	0	Non-relevant	0
ϕ_5	1741	kernel	1	1	Relevant	1
ϕ_6	2094	driver	0	1	Relevant	1
ϕ_7	3191	driver	1	0	Non-relevant	0

Training examples

s_T	s_B	Score
0	0	0
0	1	$1 - g$
1	0	g
1	1	1

Four possible combinations of s_T and s_B and the corresponding

$$\text{score}(d, q) = g * s_T(d, q) + (1 - g) * s_B(d, q)$$

Learning weights (simple machine learning)

Squared error of the scoring function with weight g on example ϕ is

$\varepsilon(g, \phi) = (\ r(d, q) - \text{score}(d, q)\ )^2$

Example	d	Query	s_T	s_B	Score	r	ε	**ε *assuming* g = 0.4**
ϕ_1	37	linux	1	1	1	1	0	0
ϕ_2	37	penguin	0	1	$1 - g$	0	$(1 - g)^2$	0.36
ϕ_3	238	system	0	1	$1 - g$	1	g^2	0.16
ϕ_4	238	penguin	0	0	0	0	0	0
ϕ_5	1741	kernel	1	1	1	1	0	0
ϕ_6	2094	driver	0	1	$1 - g$	1	g^2	0.16
ϕ_7	3191	driver	1	0	g	0	g^2	0.16
							Tot_ ε	**Tot_ ε = 0.84**

s_T	s_B	Score	**Score *assuming* g = 0.4**
0	0	0	0
0	1	$1 - g$	0.6
1	0	g	0.4
1	1	1	1

- **Exercise 6.5**
- Apply Equation 6.6 to the sample training set in Figure 6.5 to estimate the best value of *g* for this sample.
- **Exercise 6.6**
- For the value of *g* estimated in Exercise 6.5, compute the weighted zone score for each (query, document) example. How do these scores relate to the relevance judgments in Figure 6.5 (quantized to 0/1)?

$$\sum_j \varepsilon(g, \Phi_j).$$

Equation 6.4

Let n_{01r} (respectively, n_{01n}) denote the number of training examples for which $s_T(d_j, q_j) = 0$ and $s_B(d_j, q_j) = 1$ and the editorial judgment is *Relevant* (respectively, *Non-relevant*). Then the contribution to the total error in Equation (6.4) from training examples for which $s_T(d_j, q_j) = 0$ and $s_B(d_j, q_j) = 1$

is

$$[1 - (1 - g)]^2 n_{01r} + [0 - (1 - g)]^2 n_{01n}.$$

Learning weights (simple machine learning)

- Total error of a set of training examples $\mathrm{Tot\_}\,\varepsilon = \sum_j \varepsilon(g, \phi_j) = \sum_j (\, r(d_j, q) - \mathrm{score}(d_j, q)\,)^2$
- Goal is to choose g to minimize the total error.
- Note: Our example has only two zones with weights g and 1 – g, respectively! Generally, there will be *l* zones with weights $g_1, \ldots, g_l$. Same principles!

s_T	s_B	**Score**	r	***No.***	ε
0	0	0	0	n_{00n}	0
0	0	0	1	n_{00r}	1
0	1	$1-g$	0	n_{01n}	$(1-g)^2$
0	1	$1-g$	1	n_{01r}	g^2
1	0	g	0	n_{10n}	g^2
1	0	g	1	n_{10r}	$(1-g)^2$
1	1	1	0	n_{11n}	1
1	1	1	1	n_{11r}	0

Total error Tot_ ε : $(n_{01r} + n_{10n})g^2 + (n_{10r} + n_{01n})(1-g)^2 + n_{00r} + n_{11n}$

Learning weights (simple machine learning)

- Want to minimize total error **Tot_ ε** $= (n_{01r} + n_{10n})g^2 + (n_{10r} + n_{01n})(1 - g)^2 + n_{00r} + n_{11n}$
- Differentiating w.r.t. g: d(Tot_ ε)/dg

$$= 2(n_{01r} + n_{10n})g - 2(n_{10r} + n_{01n})(1 - g)$$

- Find minimum by solving:

$$2(n_{01r} + n_{10n})g - 2(n_{10r} + n_{01n})(1 - g) = 0$$

$$\rightarrow \quad (n_{10r} + n_{10n} + n_{01r} + n_{01n})g = n_{10r} + n_{01n}$$

$$\rightarrow \quad \boldsymbol{g = (n_{10r} + n_{01n}) / (n_{10r} + n_{10n} + n_{01r} + n_{01n})}$$

$$\rightarrow \quad g = (0 + 1) / (0+1+2+1) = 1/4 = 0.25$$

Learning weights (simple machine learning)

Squared error of the scoring function with weight g on example ϕ is

$\varepsilon(g, \phi) = (r(d, q) - \text{score}(d, q))^2$

Example	d	Query	s_T	s_B	Score	r	ε	ε assuming $g = 0.25$
ϕ_1	37	linux	1	1	1	1	0	0
ϕ_2	37	penguin	0	1	$1 - g$	0	$(1 - g)^2$	0.5625
ϕ_3	238	system	0	1	$1 - g$	1	g^2	0.0625
ϕ_4	238	penguin	0	0	0	0	0	0
ϕ_5	1741	kernel	1	1	1	1	0	0
ϕ_6	2094	driver	0	1	$1 - g$	1	g^2	0.0625
ϕ_7	3191	driver	1	0	g	0	g^2	0.0625
							Tot_ ε	Tot_ ε = 0.75

s_T	s_B	Score	Score assuming $g = 0.25$
0	0	0	0
0	1	$1 - g$	0.75
1	0	g	0.25
1	1	1	1

Term Frequency and Weighing

- Scoring has hinged on whether or not a query term is present in a zone within a document.
- Next logical step: a document or zone that mentions a query term more often has more to do with that query and therefore should receive a higher score.
- We assign to each term in a document a *weight* for that term, that depends on the number of occurrences of the term in the document.
- We would like to compute a score between a query term *t* and a document *d*, based on the weight of *t* in *d*.
- The simplest approach is to assign the weight to be equal to the number of occurrences of term *t* in document *d*.
- This weighting scheme is referred to as TERM FREQUENCY and is denoted $\mathrm{tf}_{t,d,}$ with the subscripts denoting the term and the document in order.

Recall: Binary term-document incidence matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
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

Each document is represented by a binary vector $\in \{0,1\}^{|V|}$

Term-document count matrices

- Consider the number of occurrences of a term in a document:
  - Each document is a $\boxed{\text{count vector}}$ in $\mathbb{N}^{v}$: a column below

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	157	73	0	0	0	0
Brutus	4	157	0	1	0	0
Caesar	232	227	0	2	1	1
Calpurnia	0	10	0	0	0	0
Cleopatra	57	0	0	0	0	0
mercy	2	0	3	5	5	1
worser	2	0	1	1	1	0

Bag of words model

- For a document *d*, the set of weights determined by the tf weights above may be viewed as a quantitative digest of that document.
- *Bag of words model,* the exact ordering of the terms in a document is ignored but the number of occurrences of each term is material.
- In *John is quicker than Mary* and *Mary is quicker than John* have the same scores
- This is called the bag of words model.
- It seems intuitive that two documents with similar bag of words representations are similar in content.
- But, in a sense, this is a step back: The positional index was able to distinguish these two documents.
- Are all words in a document equally important?

Term frequency tf

- The term frequency $\text{tf}_{t,d}$ of term *t* in document *d* is defined as the number of times that *t* occurs in *d*.
- We want to use tf when computing query-document match scores. But how?
- Raw term frequency- all terms are considered equally important when it comes to assessing relevancy on a query.
- Raw term frequency is not what we want:
  - A document with 10 occurrences of the term is more relevant than a document with 1 occurrence of the term.
  - But not 10 times more relevant.
- Relevance does not increase proportionally with term frequency.

Document frequency

- Rare terms are more informative than frequent terms
  - Recall stop words
- Consider a term in the query that is rare in the collection (e.g., *arachnocentric*)
- A document containing this term is very likely to be relevant to the query *arachnocentric*
- → We want a high weight for rare terms like *arachnocentric*.

Document frequency, continued

- Frequent terms are less informative than rare terms
- Consider a query term that is frequent in the collection (e.g., *high, increase, line*)
- A document containing such a term is more likely to be relevant than a document that doesn't
- But it's not a sure indicator of relevance.
- → For frequent terms, we want positive weights for words like *high, increase, and line*
- But lower weights than for rare terms.
- Therefore, there is need to attenuate the effect of terms that occur too often in the collection for relevance determination.
- We will use document frequency (df) to capture this.
- *Document frequency* dft, is defined to be the number of documents in the collection that contain a term *t*.

Collection vs. Document frequency

- The collection frequency of *t* is the number of occurrences of *t* in the collection, counting multiple occurrences.
  - The idea would be to reduce the tf weight of a term by a factor that grows with its collection frequency.
- Example:

Word	Collection frequency	Document frequency
insurance	10440	3997
try	10422	8760

- Which word is a better search term (and should get a higher weight)?
  - For the purpose of scoring it is better to use a document-level statistic

Inverse document frequency - idf

- How is the document frequency df of a term used to scale its weight?
- Denoting the total number of documents in a collection as *N*, we define the *inverse document frequency* (idf) of a term *t* as follows:

$$\mathrm{idf}_t = \log_{10}\left(N/\mathrm{df}_t\right)$$

- df_t is the document frequency of *t*: the number of documents that contain *t*
- We use log (N/df_t) instead of N/df_t to "dampen" the effect of idf.

idf example, suppose N = 1 million

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1,000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

$$\mathrm{idf}_t = \log_{10}\,(N/\mathrm{df}_t)$$

There is one idf value for each term t in a collection.

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

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low.

Effect of idf on ranking

- Does idf have an effect on ranking for one-term queries, like
  - iPhone
- idf has no effect on ranking one term queries
  - idf affects the ranking of documents for queries with at least two terms
  - For the query capricious person, idf weighting makes occurrences of capricious count for much more in the final document ranking than occurrences of person.

tf-idf weighting

- The definitions of term frequency and inverse document frequency are combined to produce a composite weight for each term in each document.
- The tf-idf weight of a term is the product of its tf weight and its idf weight.

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

- Best known weighting scheme in information retrieval
  - Note: the "-" in tf-idf is a hyphen, not a minus sign!
  - Alternative names: tf.idf, tf x idf

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.

Score for a document given a query

- *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*.

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

Binary → count → weight matrix

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	5.25	3.18	0	0	0	0.35
Brutus	1.21	6.1	0	1	0	0
Caesar	8.59	2.54	0	1.51	0.25	0
Calpurnia	0	1.54	0	0	0	0
Cleopatra	2.85	0	0	0	0	0
mercy	1.51	0	1.9	0.12	5.25	0.88
worser	1.37	0	0.11	4.15	0.25	1.95

Each document is now represented by a real-valued vector of tf-idf weights $\in \mathbb{R}^{|V|}$

Documents as vectors

- 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 tf-idf $_{t,d}$
- So we have a |V|-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional space: tens of millions of dimensions in case of a web search engine
- These are very sparse vectors - most entries are zero.

Consider the table of term frequencies for 3 documents denoted Doc1, Doc2, Doc3 in Figure 6.9. Compute the tf-idf weights for the terms car, auto, insuranc best, for each document, using the idf values from Figure 6.8.

Solution :

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

	Doc1	Doc2	Doc3
car	27	4	24
auto	3	33	0
insurance	0	33	29
best	14	0	17

► **Figure 6.9** Table of tf values for Exercise 6.10.

tf-idf = tf * idf

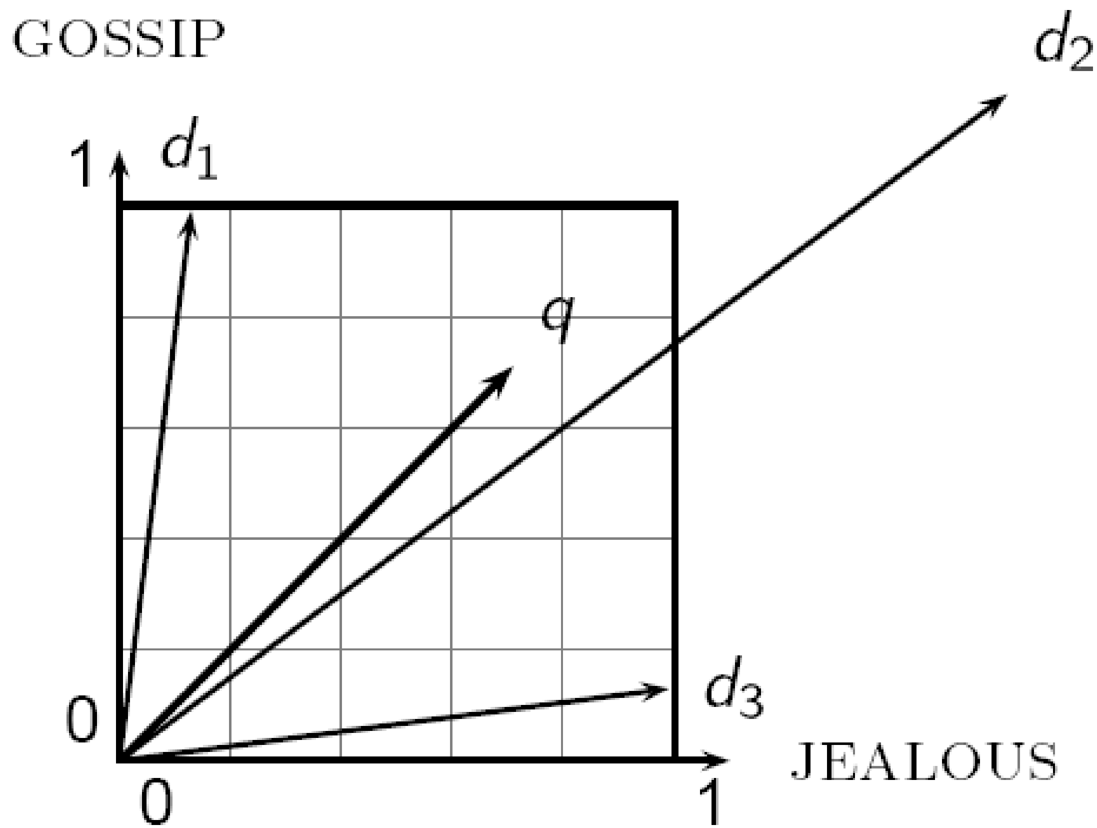
terms	Doc1	Doc2	Doc3
Car	44.55	6.6	39.6
Auto	6.24	68.64	0
Insurance	0	53.46	46.98
Best	21	0	25.5

Formalizing vector space proximity

- First cut: distance between two points
  - (= distance between the end points of the two vectors)
- Euclidean distance?
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is large for vectors of different lengths.
- Two documents having similar content can have large Euclidean distance simply because one document is much longer than the other

Why distance is a bad idea

The Euclidean distance between $\vec{q}$ and $\vec{d_2}$ is large even though the distribution of terms in the query $\vec{q}$ and the distribution of terms in the document $\vec{d_2}$ are very similar.



Use angle instead of distance

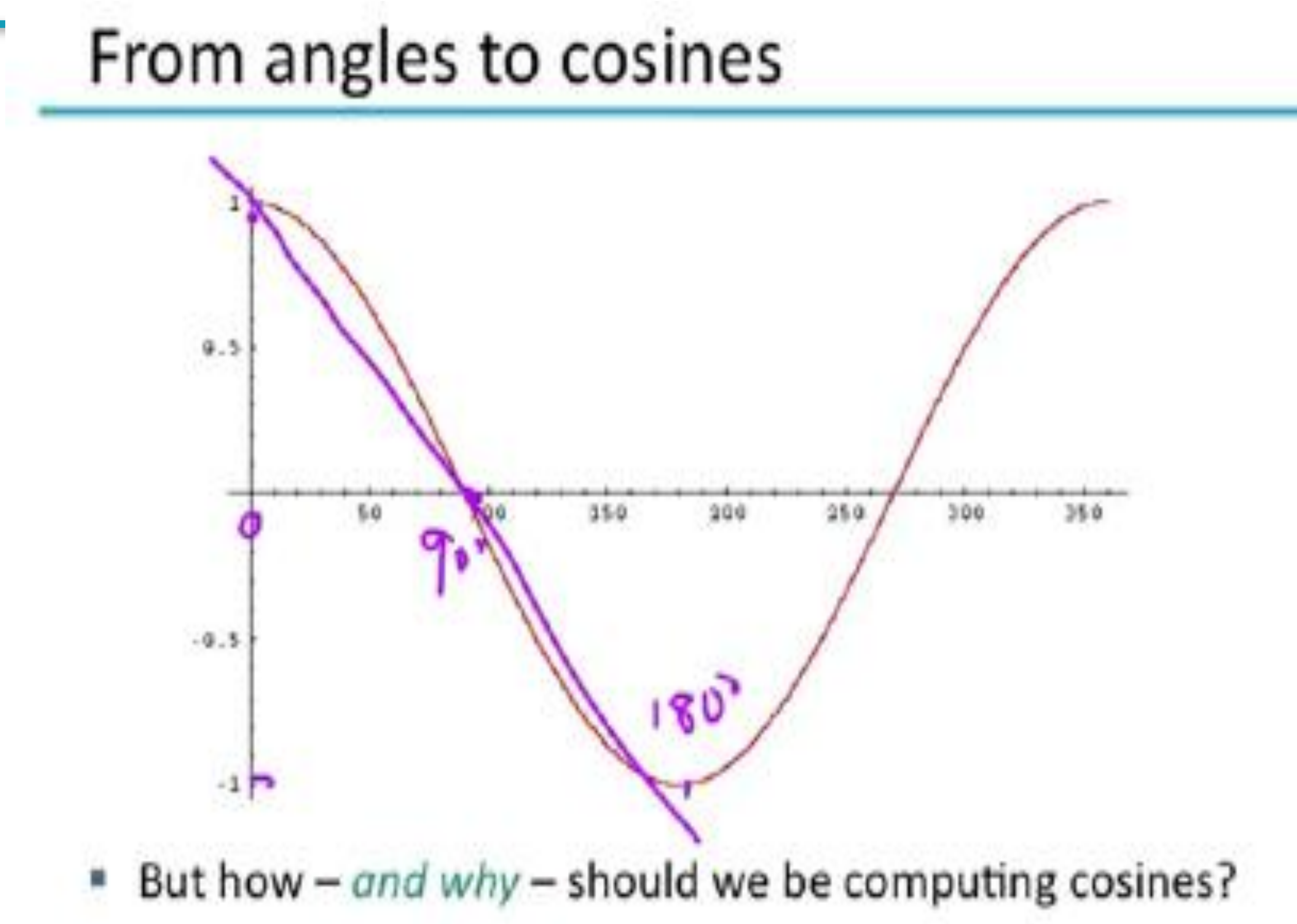
- Thought experiment: take a document *d* and append it to itself. Call this document *d*′.
- "Semantically" d and d′ have the same content
- The Euclidean distance between the two documents can be quite large
- The angle between the two documents is 0, corresponding to maximal similarity.

- Key idea: Rank documents according to angle with query.

From angles to cosines

- The following two notions are equivalent.
  - Rank documents in <u>increasing</u> order of the angle between query and document
  - Rank documents in <u>decreasing</u> order of cosine(query,document)
- Cosine is a monotonically decreasing function for the interval [0^o, 180^o]

Cosine is a monotonically decreasing function



Cosine is a monotonically decreasing function

cosine(query,document)

Dot product   Unit vectors

$$\cos(\vec{q},\vec{d}) = \frac{\vec{q} \bullet \vec{d}}{|\vec{q}||\vec{d}|} = \frac{\vec{q}}{|\vec{q}|} \bullet \frac{\vec{d}}{|\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

q_i is the tf-idf weight of term *i* in the query
d_i is the tf-idf weight of term *i* in the document

$\cos(\vec{q},\vec{d})$ is the cosine similarity of $\vec{q}$ and $\vec{d}$ … or, equivalently, the cosine of the angle between $\vec{q}$ and $\vec{d}$.

Cosine for length-normalized vectors

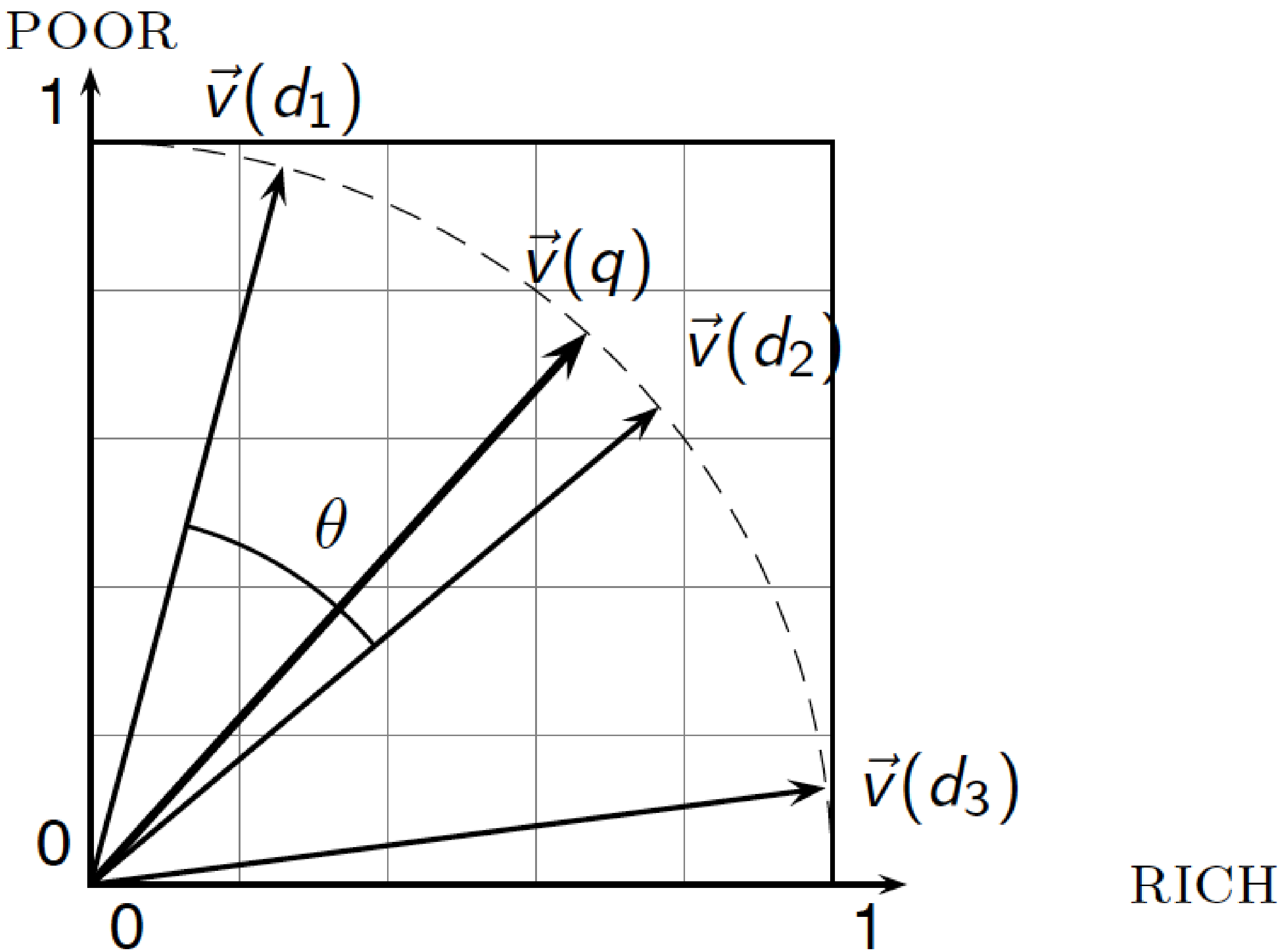
- For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

$$\cos(\vec{q},\vec{d}) = \vec{q} \bullet \vec{d} = \sum_{i=1}^{|V|} q_i d_i$$

for q, d length-normalized.

This is helpful when the direction of the vector is meaningful but the magnitude is not

Cosine similarity illustrated



Computing vector scores

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 $Scores[]$

Variant tf-idf functions

- For assigning a weight for each term in each document, a number of alternatives to tf and tf-idf have been considered.

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

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

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

tf-idf example: lnc.ltc

Document: *car insurance auto insurance*
Query: *best car insurance*

Term	Query						Document				Prod
	tf-raw	tf-wt	df	idf	wt	n'lize	tf-raw	tf-wt	wt	n'lize	
auto	0	0	5000	2.3	0	0	1	1	1	0.52	0
best	1	1	50000	1.3	1.3	0.34	0	0	0	0	0
car	1	1	10000	2.0	2.0	0.52	1	1	1	0.52	0.27
insurance	1	1	1000	3.0	3.0	0.78	2	1.3	1.3	0.68	0.53

Exercise: what is *N*, the number of docs?

Doc length $=\sqrt{1^2+0^2+1^2+1.3^2} \approx 1.92$

Score = 0+0+0.27+0.53 = 0.8

Maximum tf normalization

- We observe higher term frequencies in longer documents, merely because longer documents tend to repeat the same words over and over again.
- One well-studied technique is to normalize the tf weights of all terms occurring in a document by the maximum tf in that document.
- Normalized term frequency for each term *t* in document *d is given* by

$$\mathrm{ntf}_{t,d} = a + (1-a)\frac{\mathrm{tf}_{t,d}}{\mathrm{tf}_{\max}(d)},$$

  where *a* is a value between 0 and 1 and is generally set to 0.4, although some early work used the value 0.5. The term *a* is a *smoothing* term.
- Suppose we were to take a document *d* and create a new document *d′* by simply appending a copy of *d* to itself. While *d′* should be no more relevant to any query than *d* is, the use of

$$\mathrm{Score}(q,d) = \sum_{t \in q} \text{tf-idf}_{t,d}.$$

  would assign it twice as high a score as *d*. Replacing tf-idf $_{t,d}$ by ntf-idf $_{t,d}$ eliminates the anomaly in this example.

- Maximum tf normalization does suffer from the following issues:
  - a change in the stop word list can dramatically alter term weightings (and therefore ranking).
  - a document may contain an unusually large number of occurrences of a term, not representative of the content of that document
  - a document in which the most frequent term appears roughly as often as many other terms should be treated differently from one with a more skewed distribution.

Document and query weighting schemes
tf-idf weighting has many variants

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)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

Columns headed ‘n’ are acronyms for weight schemes.

Weighting may differ in queries vs documents

- Many search engines allow for different weightings for queries vs. documents
- SMART Notation: denotes the combination in use in an engine, with the notation *ddd.qqq,* using the acronyms from the previous table
- A very standard weighting scheme is: lnc.ltc
- Document: logarithmic tf (l as first character), no idf and cosine normalization
- Query: logarithmic tf (l in leftmost column), idf (t in second column), cosine normalization …

Pivoted normalized document length

- We normalized each document vector by the Euclidean length of the vector, so that all document vectors turned into unit vectors.
- In doing so, we eliminated all information on the length of the original document; thus masking some subtleties about longer documents.
- We introduce a form of normalizing the vector representations of documents in the collection, so that the resulting "normalized" documents are not necessarily of unit length.
- Then, when we compute the dot product score between a (unit) query vector and such a normalized document, the score is skewed to account for the effect of document length on relevance.
- This form of compensation for document length is known as *pivoted document length normalization*.

- Suppose that we were given, for each query *q* and for each document *d*, a Boolean judgment of whether or not *d* is relevant to the query *q*; we may compute a *probability of relevance* as a function of document length, averaged over all queries in the ensemble. The resulting plot may look like the curve drawn in thick lines
  - To compute this curve, we bucket documents by length and compute the fraction of relevant documents in each bucket, then plot this fraction against the median document length of each bucket.
  - On the other hand, the curve in thin lines shows the same documents and query ensemble if we were to use relevance as prescribed by cosine normalization – thus, cosine normalization
- The thin and thick curves crossover at a point *p* corresponding to document length ℓ_p, which we refer to as the *pivot length*;

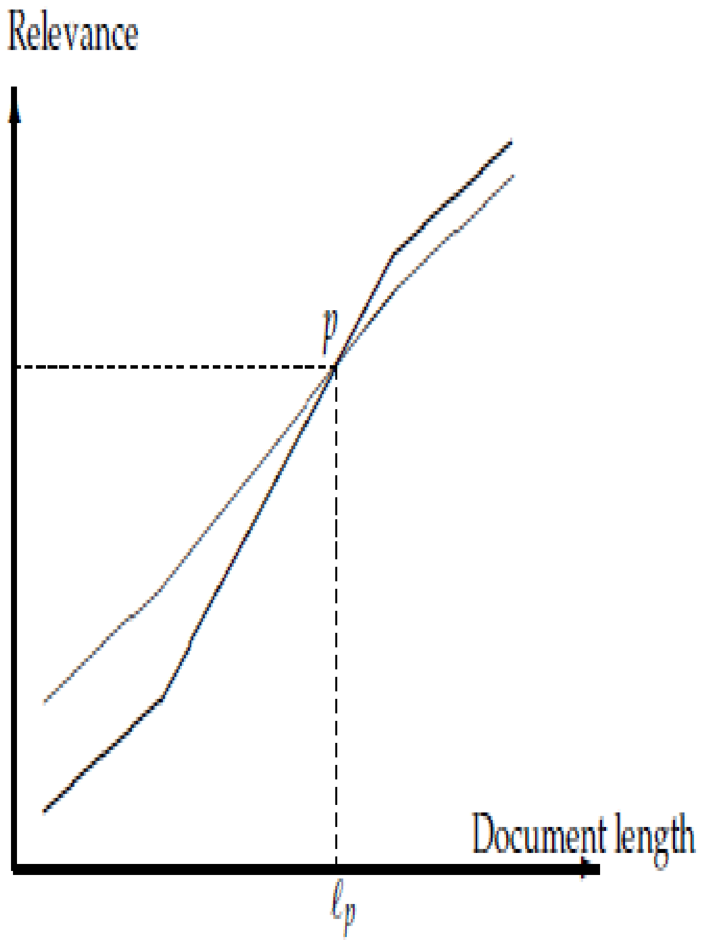
- Next "rotate" the cosine normalization curve counter-clockwise about *p* so that it more closely matches thick line representing the relevance vs. document length curve.
  - we do so by using in Equation

$$\text{score}(q,d) = \frac{\vec{V}(q) \cdot \vec{V}(d)}{|\vec{V}(q)||\vec{V}(d)|}.$$

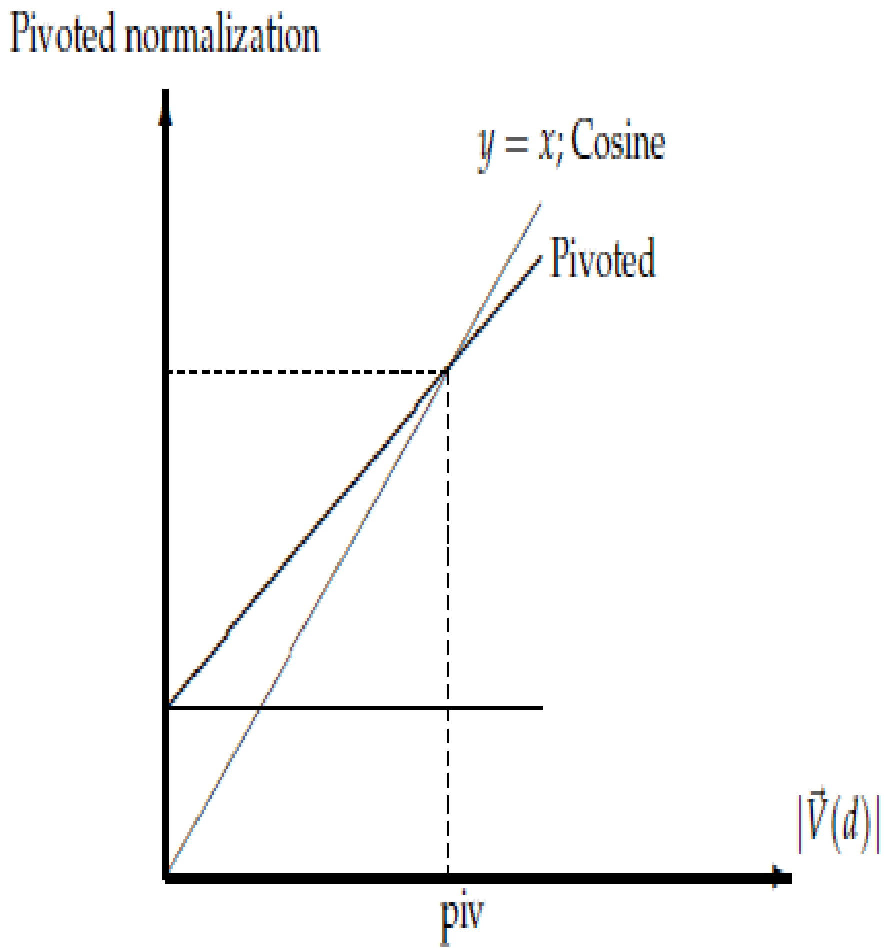
  - a normalization factor for each document vector $\vec{V}(d)$ that is not the Euclidean length of that vector, but instead one that is larger than the Euclidean length for documents of length less than ℓ*p*, and smaller for longer documents.
  - Pivoted length normalization

$$a u_d + (1-a)\text{piv},$$

  - Where a is the slope and u_d is the number of unique terms in document *d*.



▸ **Figure 6.16** Pivoted document length normalization.



▸ **Figure 6.17** Implementing pivoted document length normalization by linear scaling.

Summary – vector space ranking

- Represent the query as a weighted tf-idf vector
- Represent each document as a weighted tf vector
- Compute the cosine similarity score for the query vector and each document vector
- Rank documents with respect to the query by score
- Return the top *K* (e.g., *K* = 10) to the user