Applied Databases

Lecture 11
TFIDF Scoring, Lucene

Sebastian Maneth

University of Edinburgh - February 26th, 2017

Outline

1. Vector Space Ranking & TFIDF

2. Lucene

Next Lecture → Assignment 1 marking
will be discussed

1. Vector Space Ranking

→ quick recap of last lecture's topic,
using David Kauchak's slides

2. Vector Space Ranking

→ Represent the query as a weighted *TFIDF vector*

→ Represent each document as a weighted *TFIDF vector*

→ Compute the *cosine similarity score* between query vector and each document vector

→ Rank documents by their *score*

→ Return the top K (e.g., K = 10) documents to the user

→ slides from David Kauchak (who adapted them from Chris Manning's slides)

Term-document count matrices

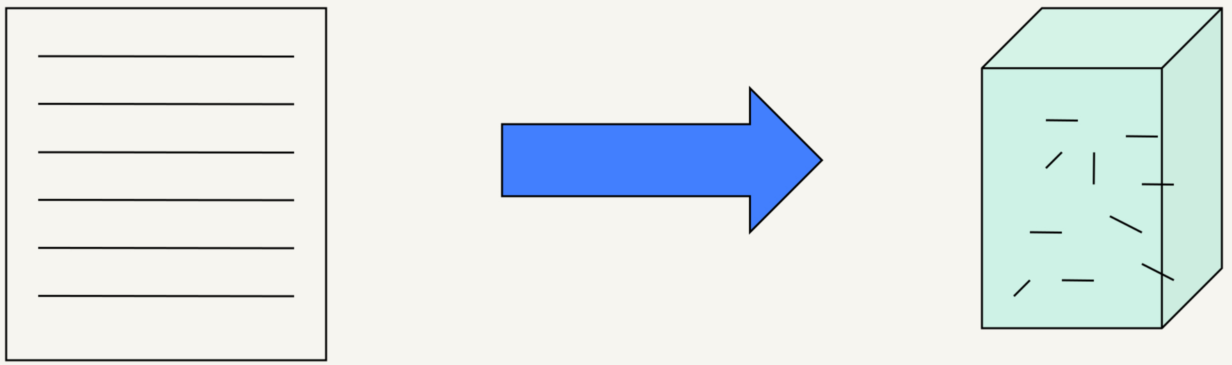
- Consider the number of occurrences of a term in a document:
  - Each document is a count vector in $\mathbb{N}^{v}$: a column below

	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

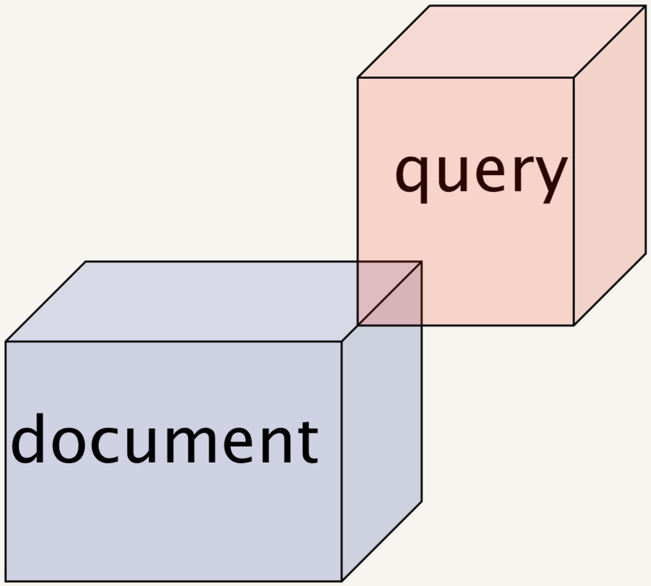
What information is lost with this representation?

Bag of words representation

- Represent a document by the occurrence counts of each word
- **Ordering** of words is lost
- *John is quicker than Mary* and *Mary is quicker than John* have the same vectors

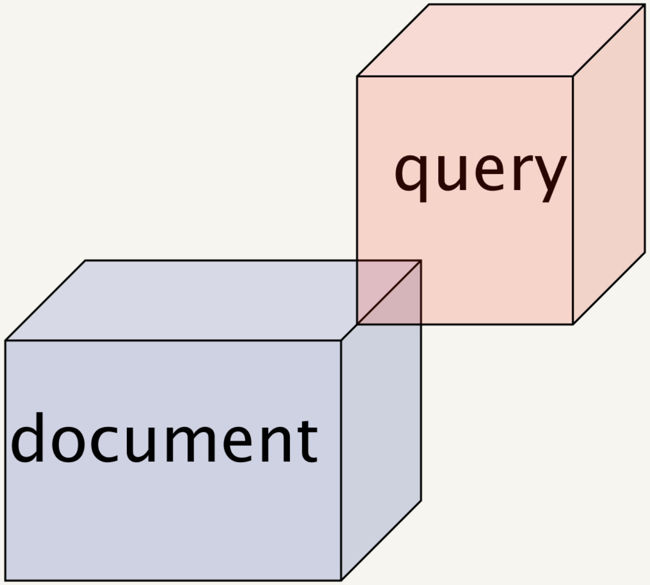


Bag of words



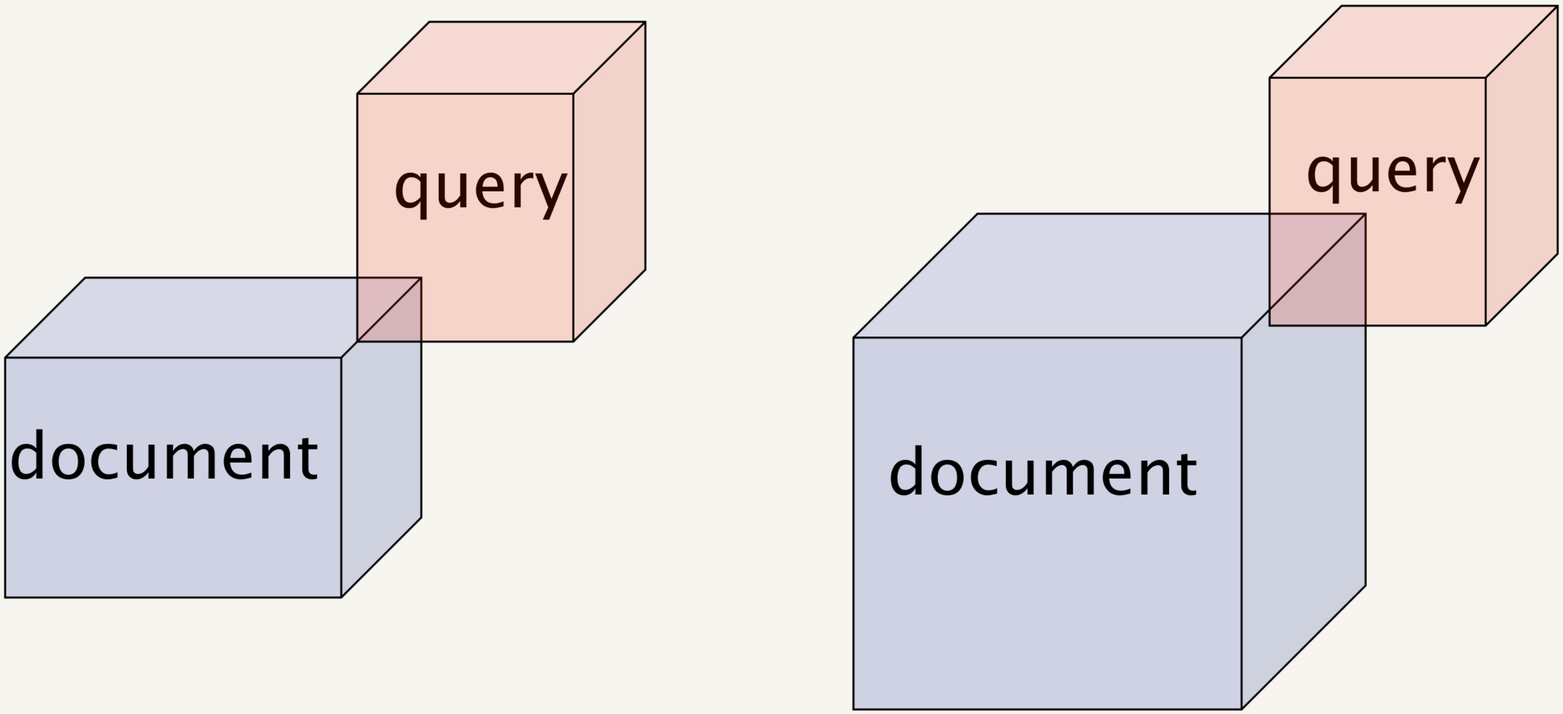
What is the notion of “intersection” for the bag or words model?

Bag of words



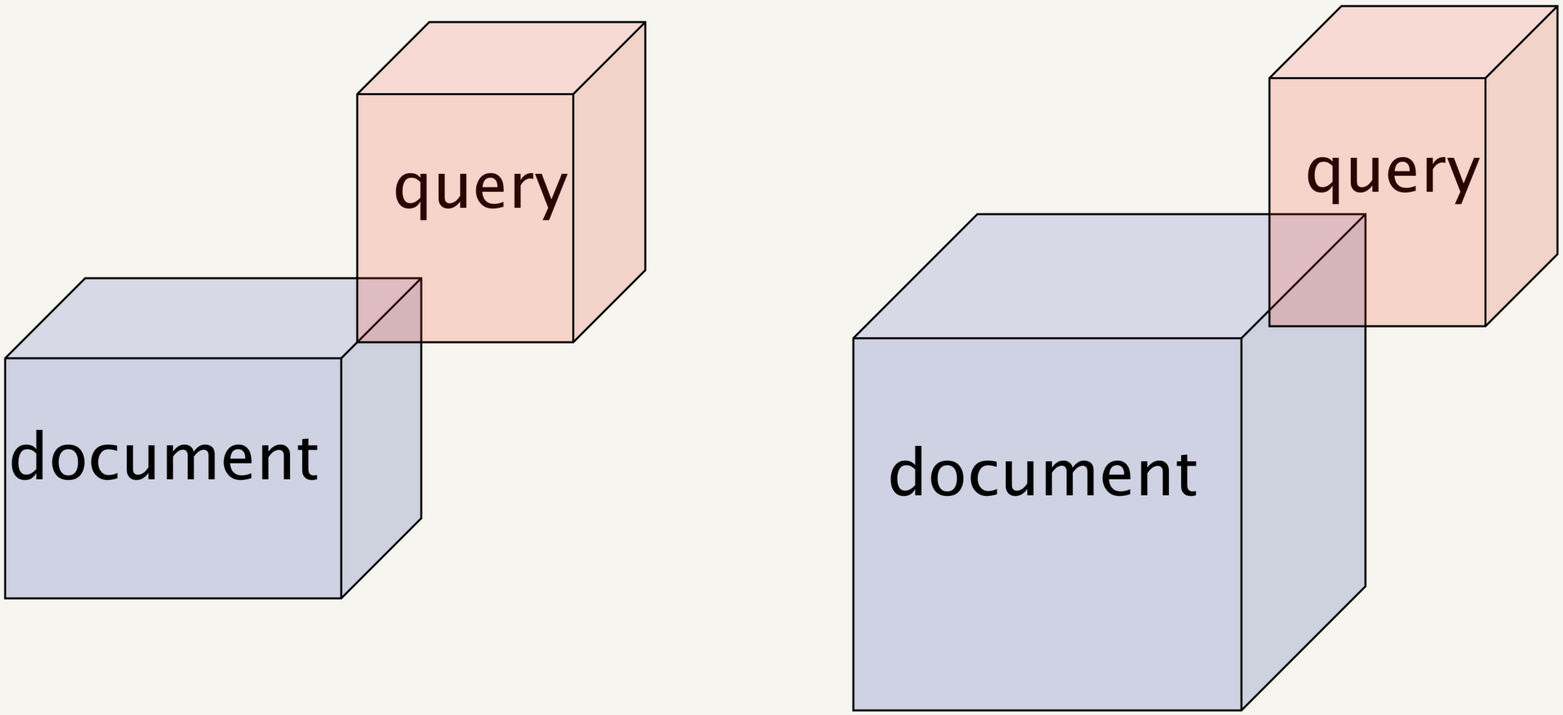
Want to take into account term frequency

Some things to be careful of…



Say I take the document and simply append it to itself. What happens to the overlap?

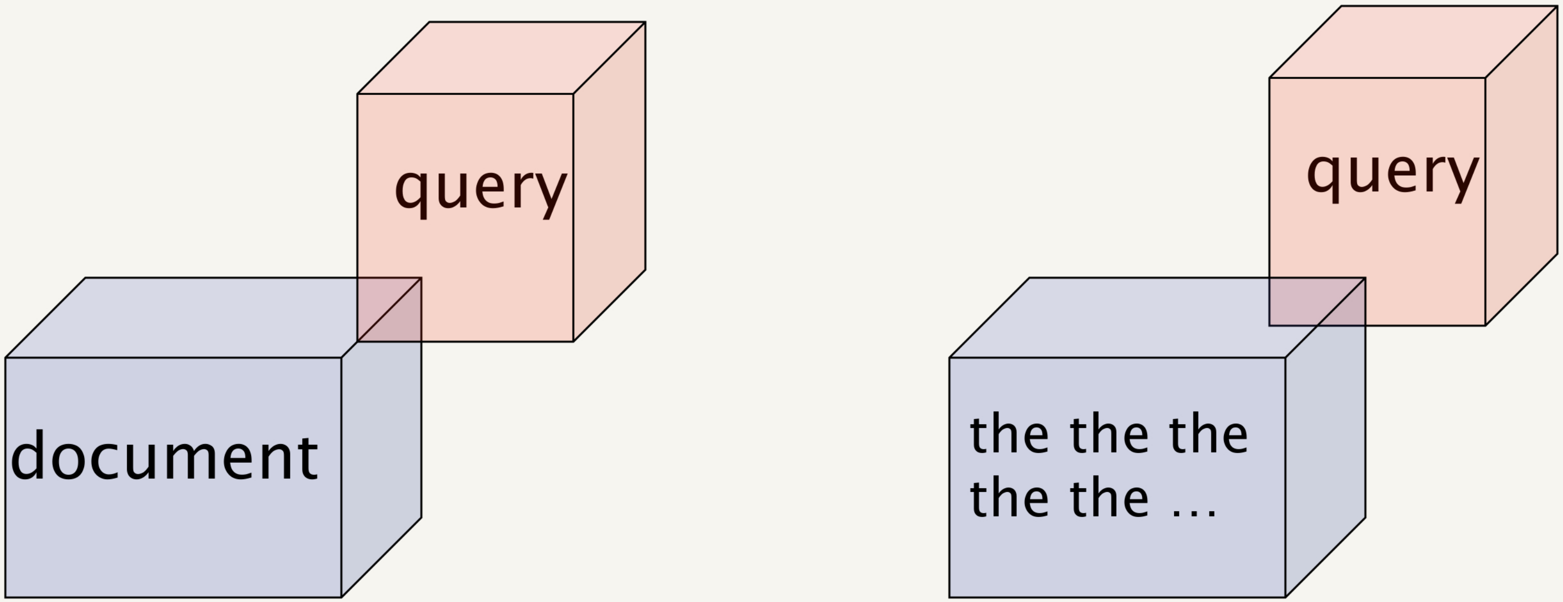
Some things to be careful of…



What is the issue?

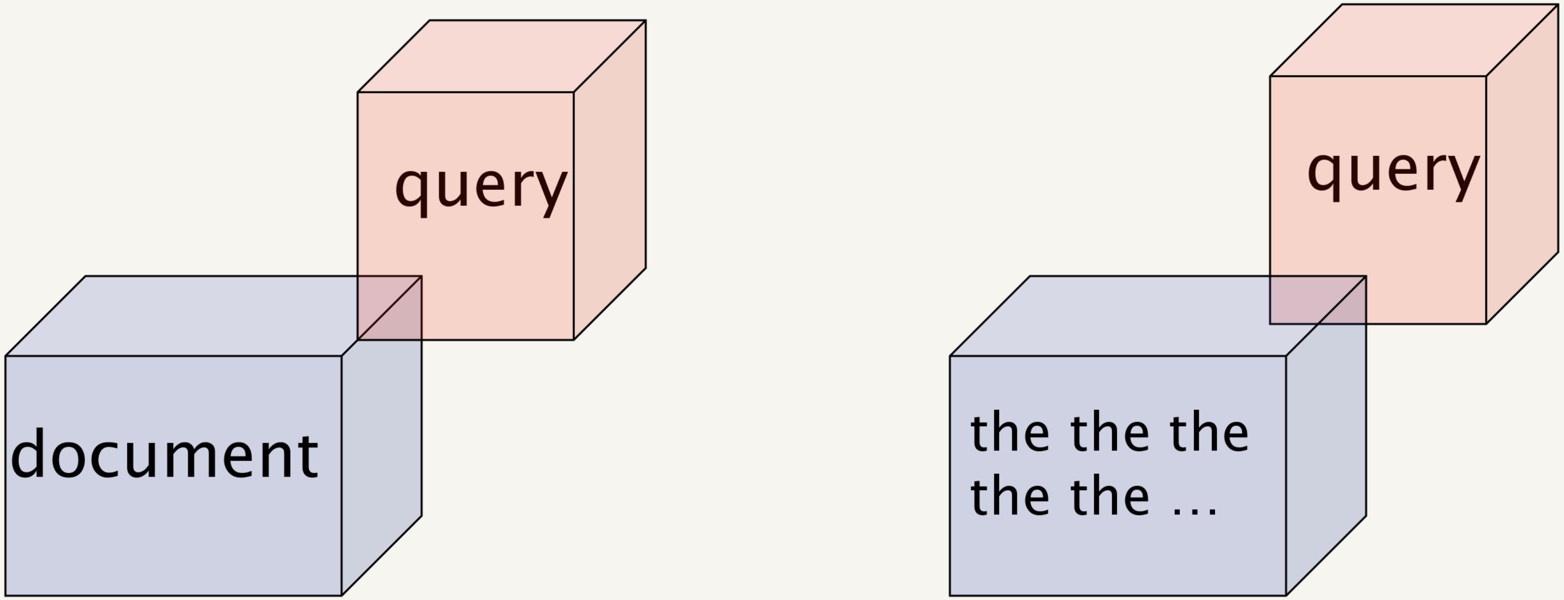
Need some notion of the length of a document

Some things to be careful of…



What about a document that contains only frequent words, e.g. the?

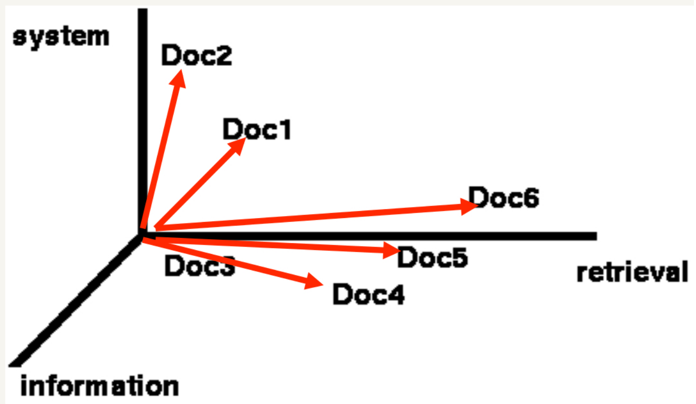
Some things to be careful of…



Need some notion of the importance of words

Documents as vectors

- We have a |V|-dimensional vector space
- Terms are axes of the space
- Documents are points or vectors in this space
- Very high-dimensional: hundreds of millions of dimensions when you apply this to a web search engine
- This is a very sparse vector - most entries are zero



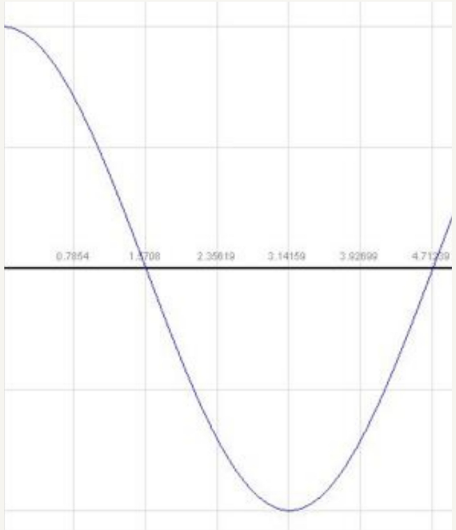
Use angle instead of distance

- 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

- Any other ideas?
- Rank documents according to angle with query

From angles to cosines

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



cosine(query,document)

Dot product

Unit vectors

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

cos(*q*,*d*) is the cosine similarity of *q* and *d* … or, equivalently, the cosine of the angle between *q* and *d*.

Inverse document frequency

- df_t is the document frequency of *t*: the number of documents that contain *t*
  - df is a measure of the informativeness of *t*
- We define the idf (inverse document frequency) of *t* by

$$\mathrm{idf}_t = \log N/\mathrm{df}_t$$

- 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

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

idf example, suppose N= 1 million

term	df_t	idf_t
calpurnia	1	
animal	100	
sunday	1,000	
fly	10,000	
under	100,000	
the	1,000,000	

What if we didn’t use the log to dampen the weighting?

idf example, suppose *N*= 1 million

term	df_t	idf_t
calpurnia	1	1,000,000
animal	100	10,000
sunday	1,000	1,000
fly	10,000	100
under	100,000	10
the	1,000,000	1

What if we didn't use the log to dampen the weighting?

Putting it all together

- We have a notion of term frequency overlap
- We have a notion of term importance
- We have a similarity measure (cosine similarity)

- Can we put all of these together?
  - Define a weighting for each term
  - The tf-idf weight of a term is the product of its tf weight and its idf weight

$$\mathrm{w}_{t,d} = \mathrm{tf}_{t,d} \times \log N / \mathrm{df}_t$$

tf-idf weighting

$$\mathrm{w}_{t,d} = \mathrm{tf}_{t,d} \times \log N / \mathrm{df}_t$$

- Best known weighting scheme in information retrieval
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection
- Works surprisingly well!
- Works in many other application domains

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 $\in \mathbb{R}^{|V|}$

We then calculate the similarity using cosine similarity with these vectors

There are many variations of TF-IDF weighting

→ log(N / DF(T))   [as on previous slides]
gives weight **zero**, to a term appearing in each document!
→ alternative log(1+ N / DF(T))

→ alternatives to TF:
- divide by largest TF of that term (“normalization”)
- take 1 + log(TF) (“log-frequencey weighting”)

There are many variations of TF-IDF weighting

→ log(N / DF(T))   [as on previous slides]
gives weight **zero**, to a term appearing in each document!
→ alternative log(1+ N / DF(T))

→ alternatives to TF:
– divide by largest TF of that term (“normalization”)
– take 1 + log(TF) (“log-frequencey weighting”)

Explanations for taking log of N / DF(T)   (“damping”)

→ Probability that *random document* contains term T:
P(T) = DF(T) / N

→ IDF(T) = – log(**P**(T))

There are many variations of TF-IDF weighting

→ log(N / DF(T)) [as on previous slides]
gives weight zero, to a term appearing in each document!
→ alternative log(1+ N / DF(T))

→ alternatives to TF: – divide by largest TF of that term ("normalization")
– take 1 + log(TF) ("log-frequencey weighting")

Explanations for taking log of N / DF(T) ("damping")

→ Probability that *random document* contains term T:
P(T) = DF(T) / N

→ IDF(T) = – log(**P**(T))

→ IDF(T1 'and' T2) = – log(**P**(T1) * **P**(T2)) = IDF(T1) + IDF(T2)

statistically independent

Recall from Information Theory:

Message probabilities p(1), p(2), p(3), …, p(N) (sum equals 1)

Information of Message m: $I(m) = – \log p(m)$

→ see Robertson's paper, linked on course web page

E.g. $p(1) = p(2) = 0.5$

$I(1) = – \log p(1) = \mathbf{1}$ ◄——— **1 bit** (because we took log-base 2)

The mathematical theory of information is based on probability theory and statistics, and measures information with several **quantities of information**. The choice of logarithmic base in the following formulae determines the unit of information entropy that is used. The most common unit of information is the bit, based on the binary logarithm. Other units include the nat, based on the natural logarithm, and the hartley, based on the base 10 or common logarithm.

Recall from Information Theory:

Message probabilities p(1), p(2), p(3), …, p(N)   (sum equals 1)

Information of Message m: $I(m) = – \log p(m)$

://en.wikipedia.org/wiki/Quantities_of_information

Self-information [edit]

Shannon derived a measure of information content called the **self-information** or **"surprisal"** of a message *m*:

$$I(m) = \log\left(\frac{1}{p(m)}\right) = -\log(p(m))$$

where $p(m) = \Pr(M = m)$ is the probability that message *m* is chosen from all possible choices in the message space M. The base of the logarithm only affects a scaling factor and, consequently, the units in which the measured information content is expressed. If the logarithm is base 2, the measure of information is expressed in units of bits.

Information is transferred from a source to a recipient only if the recipient of the information did not already have the information to begin with. Messages that convey information that is certain to happen and already known by the recipient contain no real information. Infrequently occurring messages contain more information than more frequently occurring messages. This fact is reflected in the above equation - a certain message, i.e. of probability 1, has an information measure of zero. In addition, a compound message of two (or more) unrelated (or mutually independent) messages would have a quantity of information that is the sum of the measures of information of each message individually. That fact is also reflected in the above equation, supporting the validity of its derivation.

An example: The weather forecast broadcast is: "Tonight's forecast: Dark. Continued darkness until widely scattered light in the morning." This message contains almost no information. However, a forecast of a snowstorm would certainly contain information since such does not happen every evening. There would be an even greater amount of information in an accurate forecast of snow for a warm location, such as Miami. The amount of information in a forecast of snow for a location where it never snows (impossible event) is the highest (infinity).

Recall from Information Theory:

Message probabilities p(1), p(2), p(3), …, p(N)   (sum equals 1)

Information of Message m: $I(m) = -\log p(m)$

→ see Robertson's paper, linked on course web page

→ faint relationship to Zipf's law

Mentioned in original article introducing IDF
[Karen Spärck Jones, 1972]

Log-frequency weighting

- Want to reduce the effect of multiple occurrences of a term
- A document about "Clinton" will have "Clinton" occuring many times
- Rather than use the frequency, us the log of the frequency

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

- $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 1.3$, $10 \rightarrow 2$, $1000 \rightarrow 4$, etc.

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)	$1/\mathit{CharLength}^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1 + \log(\mathrm{tf}_{t,d})}{1 + \log(\mathrm{ave}_{t \in d}(\mathrm{tf}_{t,d}))}$				

→ how does the base of the logs influence scoring / *ranking*?

→ take document length into account
(favour shorter documents)

→ e.g. divide by square root of document length
(done by Lucene, via the “LengthNorm”)

Lucene's Scoring Function

$$score(q,d) = \sum [tf(t_d) \times idf(t) \times boost(t.field_d) \times \underline{lengthNorm(t.field_d)}] \times coord(q,d) \times qNorm(q)$$

where q is the query, d a document, t a term, and:

1. *tf* is a function of the term frequency within the document (default: $\sqrt{freq}$);
2. *idf*: Inverse document frequency of t within the whole collection (default: $\log(\frac{numDocs}{docFreq+1}) + 1$);
3. *boost* is the boosting factor, if required in the query with the "^ " operator on a given field (if not specified, set to the default field);
4. *lengthNorm*: field normalization according to the number of terms. Default: $\frac{1}{\sqrt{nbTerms}}$
5. *coord*: overlapping rate of terms of the query in the given document. Default: $\frac{overlap}{maxOverlap}$
6. *qNorm*: query normalization according to its length; it corresponds to the sum of square values of terms' weight, the global value is multiplied by each term's weight.

3. Lucene

3. Lucene

→ choose appropriate Analyzer for
- casefolding
- stemming (wrt a given language)
- stopping (wrt a given language)

→ insert documents (per "field") into a collection and generate inverted files

---

→ retrieve top-K ranked documents

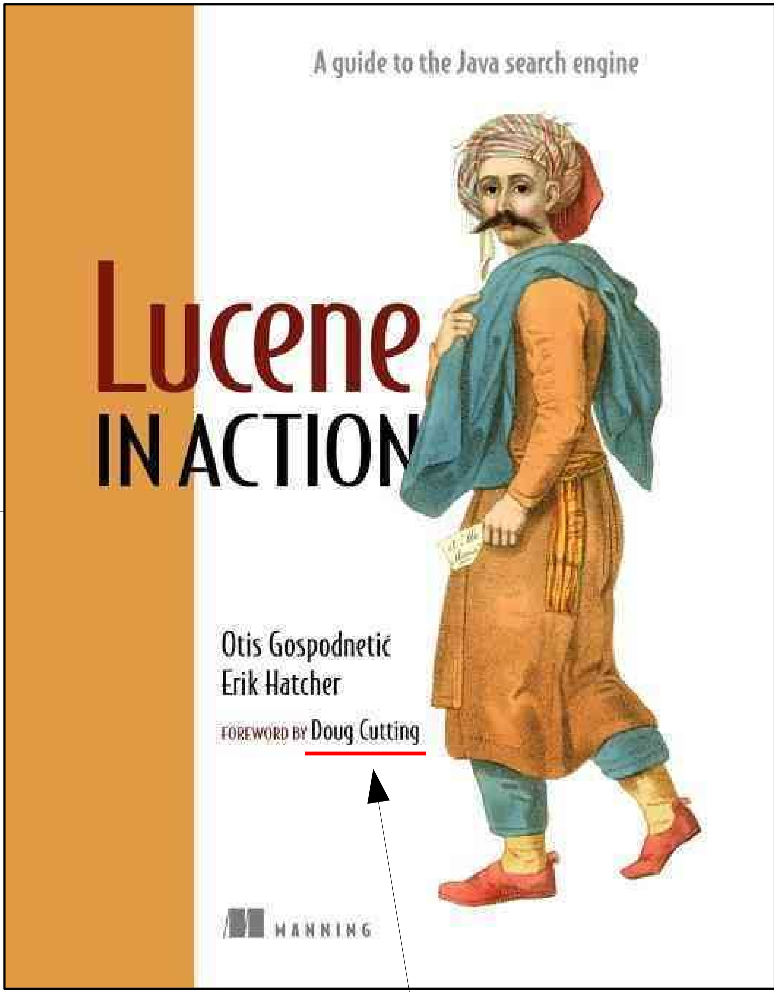
→ retrieve score of a document

3. Lucene

→ choose appropriate Analyzer for
  – casefolding
  – stemming (wrt a given language)
  – stopping (wrt a given language)

→ insert documents (per "field") and generate inverted files

→ retrieve top-K docs and their scores



1999 on SourceForge

Lucene is a huge library

→ we use Version 5.4.0

→ **most books** use older Versions, e.g. Versions 4 or 3

→ the Versions are **not** downward compatible :-(

Lucene Indexing

store original text

```
public static void insertDoc(IndexWriter i, String doc_id, String line){
  Document doc = new Document();
  doc.add(new TextField("doc_id", doc_id, Field.Store.YES));
  doc.add(new TextField("line", line,Field.Store.YES));
  try { i.addDocument(doc); } catch (Exception e) { e.printStackTrace(); }
}

public static void rebuildIndexes(String indexPath) {
  try {
    . . .
    IndexWriterConfig config=new IndexWriterConfig(new SimpleAnalyzer());
    IndexWriter i = new IndexWriter(directory, config);
    i.deleteAll();
    insertDoc(i, "1", "The old night keeper keeps the keep in the town");
    insertDoc(i, "2", "In the big old house in the big old gown.");
    . . .
```

doc_id field **line** field

Full code for indexing some documents to an index on disk (directory `indexPath`)

path name (from command line)

```
public static void insertDoc(IndexWriter i, String doc_id, String line){
  Document doc = new Document();
  doc.add(new TextField("doc_id", doc_id, Field.Store.YES));
  doc.add(new TextField("line", line,Field.Store.YES));
  try { i.addDocument(doc); } catch (Exception e) { e.printStackTrace(); }
}

public static void rebuildIndexes(String indexPath) {
  try {
    Path path = Paths.get(indexPath);
    System.out.println("Indexing to directory " + indexPath);
    Directory directory = FSDirectory.open(path);
    IndexWriterConfig config = new IndexWriterConfig(new SimpleAnalyzer());
    IndexWriter i = new IndexWriter(directory, config);
    i.deleteAll();
    insertDoc(i, "1", "The old night keeper keeps the keep in the town");
    insertDoc(i, "2", "In the big old house in the big old gown.");
    insertDoc(i, "3", "The house in the town had the big old keep");
    insertDoc(i, "4", "Where the old night keeper never did sleep.");
    insertDoc(i, "5", "The night keeper keeps the keep in the night");
    insertDoc(i, "6", "And keeps in the dark and sleeps in the light.");
    i.close();
    directory.close();
    }
  catch (Exception e) { e.printStackTrace(); }
}
```

3. Lucene

```
public static void rebuildIndexes(String indexPath) {
  try {
    . . .
    IndexWriterConfig config = new IndexWriterConfig(new SimpleAnalyzer());
    IndexWriter i = new IndexWriter(directory, config);
```

SimpleAnalyzer
→ Analyzer that filters LetterTokenizer with LowerCaseFilter

LetterTokenizer
→ divides text at non-letters.
→ tokens as maximal strings of adjacent letters,
as defined by java.lang.Character.isLetter() predicate.

Note: this does a decent job for most European languages, but does a terrible job for some Asian languages, where words are not separated by spaces.

LowerCaseFilter
→ Normalizes token text to lower case.

Keyword Search

```
Private static TopDocs search(String searchText);
  . . .
  IndexReader indexReader =  DirectoryReader.open(directory);
  IndexSearcher indexSearcher = new IndexSearcher(indexReader);
  QueryParser queryParser = new QueryParser(searchField, new SimpleAnalyzer());

  Query query = queryParser.parse(searchText);
  TopDocs topDocs = indexSearcher.search(query,10000);
  System.out.println("Number of Hits: " + topDocs.totalHits);
  for (ScoreDoc scoreDoc:topDocs.scoreDocs) {
    Document doc = indexSearcher.doc(scoreDoc.doc);
    System.out.println("doc_id: " + doc.get("doc_id")
                     + ", score: " + scoreDoc.score
                     + " [" + doc.get("line") +"]");
    }
```

Top-K (K= 10000)

Output
→ doc_id
→ score
→ content (line)

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

Fig. 1. The Keeper database. It consists of six one-line documents.

```
$ java Searcher “old”

Running search(old, line)
Number of Hits: 4
doc_id: 2, score: 0.5225172 [In the big old house in the big old gown.]
doc_id: 1, score: 0.36947548 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.36947548 [The house in the town had the big old keep]
doc_id: 4, score: 0.36947548 [Where the old night keeper never did sleep.]
```

1	`The old night keeper keeps the keep in the town`
2	`In the big old house in the big old gown.`
3	`The house in the town had the big old keep`
4	`Where the old night keeper never did sleep.`
5	`The night keeper keeps the keep in the night`
6	`And keeps in the dark and sleeps in the light.`

Fig. 1. The `Keeper` database. It consists of six one-line documents.

```
$ java Searcher "old"

Running search(old, line)
Number of Hits: 4
doc_id: 2, score: 0.5225172 [In the big old house in the big old gown.]
doc_id: 1, score: 0.36947548 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.36947548 [The house in the town had the big old keep]
doc_id: 4, score: 0.36947548 [Where the old night keeper never did sleep.]

Explanation: 0.5225172 = weight(line:old in 1) [DefaultSimilarity], result of:
  0.5225172 = fieldWeight in 1, product of:
    1.4142135 = tf(freq=2.0), with freq of:
      2.0 = termFreq=2.0
    1.1823215 = idf(docFreq=4, maxDocs=6)
    0.3125 = fieldNorm(doc=1)
```

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

Fig. 1. The Keeper database. It consists of six one-line documents.

```
$ java Searcher “big old house”

Running search(big old house, line)
Number of Hits: 4
doc_id: 2, score: 1.0412337 [In the big old house in the big old gown.]
doc_id: 3, score: 0.83452004 [The house in the town had the big old keep]
doc_id: 1, score: 0.054527204 [The old night keeper keeps the keep in the town]
doc_id: 4, score: 0.054527204 [Where the old night keeper never did sleep.]
```

```
1    The old night keeper keeps the keep in the town
2    In the big old house in the big old gown.
3    The house in the town had the big old keep
4    Where the old night keeper never did sleep.
5    The night keeper keeps the keep in the night
6    And keeps in the dark and sleeps in the light.
```

Fig. 1. The Keeper database. It consists of six one-line documents.

```
$ java Searcher “the”

Running search(the, line)
Number of Hits: 6
doc_id: 1, score: 0.4578294 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.4578294 [The house in the town had the big old keep]
doc_id: 5, score: 0.4578294 [The night keeper keeps the keep in the night]
doc_id: 2, score: 0.37381613 [In the big old house in the big old gown.]
doc_id: 6, score: 0.37381613 [And keeps in the dark and sleeps in the light.]
doc_id: 4, score: 0.2643279 [Where the old night keeper never did sleep.]
```

```
1     The old night keeper keeps the keep in the town
2     In the big old house in the big old gown.
3     The house in the town had the big old keep
4     Where the old night keeper never did sleep.
5     The night keeper keeps the keep in the night
6     And keeps in the dark and sleeps in the light.
```

```
7     The house is the house.
8     The house.
```

Even shorter

shorter

```
$ java Searcher "the"

Running search(the, line)
Number of Hits: 8
doc_id: 8, score: 0.55138564 [The house.]
doc_id: 7, score: 0.5458439 [The house is the house.]
doc_id: 1, score: 0.47751394 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.47751394 [The house in the town had the big old keep]
doc_id: 5, score: 0.47751394 [The night keeper keeps the keep in the night]
doc_id: 2, score: 0.38988853 [In the big old house in the big old gown.]
doc_id: 6, score: 0.38988853 [And keeps in the dark and sleeps in the light.]
doc_id: 4, score: 0.27569282 [Where the old night keeper never did sleep.]
```

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

7 **The house is the house.**
8 **The house.**
9 **the-the_the__the.**
10 **the-the___the.**
11 **the-thethe__the.**
12 **The.**
13 **The.**
14 **The a b c.**
15 **The a b.**
16 **The a.**

```
$ java Searcher “the”
doc_id: 9, score: 0.9393754 [the-the_the__the.]
doc_id: 12, score: 0.9393754 [The.]
doc_id: 13, score: 0.83029836 [The the.]
doc_id: 10, score: 0.81352293 [the-the___the.]
doc_id: 11, score: 0.6642387 [the-thethe__the.]
doc_id: 8, score: 0.5871096 [The house.]
doc_id: 16, score: 0.5871096 [The a.]
doc_id: 7, score: 0.5812088 [The house is the house.]
doc_id: 1, score: 0.5084518 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.5084518 [The house in the town had the big old keep]
doc_id: 5, score: 0.5084518 [The night keeper keeps the keep in the night]
doc_id: 14, score: 0.4696877 [The a b c.]
doc_id: 15, score: 0.4696877 [The a b.]
doc_id: 2, score: 0.41514918 [In the big old house in the big old gown.]
doc_id: 6, score: 0.41514918 [And keeps in the dark and sleeps in the light.]
doc_id: 4, score: 0.2935548 [Where the old night keeper never did sleep.]
```

1	`The old night keeper keeps the keep in the town`
2	`In the big old house in the big old gown.`
3	`The house in the town had the big old keep`
4	`Where the old night keeper never did sleep.`
5	`The night keeper keeps the keep in the night`
6	`And keeps in the dark and sleeps in the light.`

SimpleAnalyzer

→ filters LetterTokenizer with LowerCaseFilter

StandardAnalyzer

→ filters StandardTokenizer with StandardFilter, LowerCaseFilter and StopFilter, using a list of English stop words.

StandardTokenizer

→ grammar-based tokenizer (done in JFlex), implements the Word Break rules from the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29.

Standard Filter

→ normalizes tokens extracted with StandardTokenizer.

StopFilter

→ removes stop words from a token stream.

StandardAnalyzer – Search

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

```
$ java Searcher "the"

Running search(the, line)
Number of Hits: 0
```

StandardAnalyzer – Search

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

```
$ java Searcher “the”

Running search(the, line)
Number of Hits: 0

$ java Searcher “and”

Running search(and, line)
Number of Hits: 0
```

StandardAnalyzer – Search

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

```
$ java Searcher "the"

Running search(the, line)
Number of Hits: 0

$ java Searcher "and"

Running search(and, line)
Number of Hits: 0

$ java Searcher "in"

Running search(in, line)
Number of Hits: 0
```

StandardAnalyzer – Search

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

```
$ java Searcher "keeper"

Running search(keeper, line)
Number of Hits: 3
doc_id: 5, score: 0.614891 [The night keeper keeps the keep in the night]
doc_id: 1, score: 0.5270494 [The old night keeper keeps the keep in the town]
doc_id: 4, score: 0.5270494 [Where the old night keeper never did sleep.]
```

StandardAnalyzer – Search

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

```
$ java Searcher “keeping”

Running search(keeping, line)
Number of Hits: 0
```

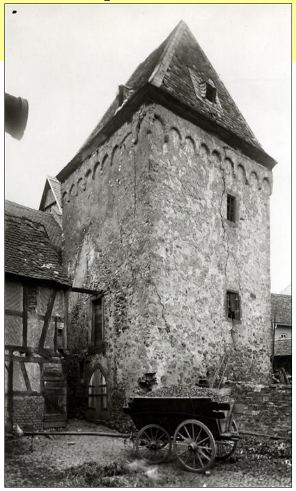
→ stemming?

EnglishAnalyzer

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

```
$ java Searcher "keeping"

Running search(keeping, line)
Number of Hits: 3
doc_id: 5, score: 0.614891 [The night keeper keeps the keep in the night]
doc_id: 1, score: 0.5270494 [The old night keeper keeps the keep in the town]
doc_id: 3, score: 0.5270494 [The house in the town had the big old keep]
```



Stemming

→ EnglishAnalyzer (in the Query part)

A reconstruction of York Castle in the 14th century, showing the castle's stone keep (top) overlooking the castle bailey (below)

END

Lecture 11