A brief introduction to Information Retrieval

Mark Johnson

Department of Computing
Macquarie University

Readings for today's talk

- *Natural Language Processing: Analyzing Text with Python and the Natural Language Toolkit*
  - Steven Bird, Ewan Klein, and Edward Loper
  - The book describing NLTK
  - `http://www.nltk.org/book`
- *Introduction to Information Retrieval*
  - Manning, Raghavan and Schütze.
  - Cambridge University Press. 2008. ISBN: 0521865719.
  - `http://nlp.stanford.edu/IR-book/`

Machine learning and data mining

- Huge amounts of data are now on-line
  - much of it is *unstructured text*
- *Data mining:* extracting information from large data sources
  - *Big data:* the data is so large that standard techniques (hardware, algorithms, etc.) cannot be used
- *Machine learning:* techniques for generalising from data
  - *Supervised learning:* data comes with *labels*, goal is to *generalise to new data*
    - identify stock take-over announcements in financial news
    - choosing most profitable ads to display on web pages
    - identify autistic children from their brain scans
  - *Unsupervised learning:* goal is to group or *cluster* data in meaningful ways
    - detecting and tracking *topics* in news or social media
    - find the *translations of words* in parallel corpora
    - identify different kinds of customers for market research

MACQUARIE UNIVERSITY

Outline

MACQUARIE UNIVERSITY

Information retrieval terminology

Document
- A unit of text available for retrieval

Collection
- A set of documents used for retrieval

Term
- The elements of documents used for retrieval
- Usually words or phrases

Query
- A user's information need expressed using terms

Diversity of information retrieval applications

- Web search engines:
  - large number of web pages
    - highly variable
    - constantly changing
  - must be *easy to use*
  - many web pages about most topics (*redundancy*)
    - ⇒ don't need to retrieve all relevant documents
    - ⇒ sort documents by relevance, i.e., *ranked retrieval*
- Specialised document retrieval, e.g., *law records*
  - high quality *manually curated* collections *with metadata*
  - highly-trained users (e.g., legal librarians)
    - can use specialised *query languages*
  - very important to retrieve all relevant documents

Precision and recall

- Precision and recall are two ways of *measuring the accuracy of an IR system*
- Suppose an IR system returns a set S of documents for some query, but we know the correct or "gold" set of documents for that query is G:
  - the *correct documents the system returned* is $C = S \cap G$
  - *recall* is the *fraction of gold documents that the system finds*

$$\textit{recall} = \frac{|S \cap G|}{|G|} = \frac{|C|}{|G|} \tag{1}$$

  - *precision* is the *fraction of documents that the system returns that are correct*

$$\textit{precision} = \frac{|S \cap G|}{|S|} = \frac{|C|}{|S|} \tag{2}$$

MACQUARIE UNIVERSITY

Precision and recall example

- *Document collection:* {'Anthony and Cleopatra', 'Julius Caesar', 'The Tempest', 'Hamlet', 'Macbeth'}
- *Query:* which documents mention *Brutus*?
- *System answer:*
  $S = \{$'Julius Caesar', 'The Tempest', 'Hamlet', 'Macbeth'$\}$
- *Gold answer:* $G = \{$'Anthony and Cleopatra', 'Julius Caesar', 'Hamlet'$\}$
- $C = S \cap G = \{$'Julius Caesar', 'Hamlet'$\}$
- *recall* $= |C|/|G| = 2/3$, i.e., system found 2/3 of correct docs
- *precision* $= |C|/|S| = 2/4$, i.e., 1/2 of system's answer was correct

MACQUARIE UNIVERSITY

The precision/recall tradeoff

- A trivial algorithm can achieve perfect recall *(how?)*
- It's often easy to achieve very high precision *(how?)*
- Often IR algorithms can be *tuned* to *optimise either precision or recall*
- Precision is usually more important than recall if:
  - the same information is in many documents (*redundancy*)
  - the user is not prepared to look through many documents
- Recall is usually more important than precision if:
  - a valuable piece of information might be in a single document
  - the user is prepared to inspect many documents

More advanced accuracy measures

- Often desirable to have a single measure of system accuracy
- *F-score* is the *harmonic mean of precision and recall*

$$\textit{f-score} \quad = \quad \frac{1}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} \; = \; \frac{2\,|C|}{|S| + |G|}$$

- In a real information retrieval application, it's impossible to find all the gold documents $G \Rightarrow$ can't calculate recall
  - we can calculate precision by manually scoring system output
- *Mean average precision* (MAP) is precision averaged over
  - several different queries
  - many different levels of recall

Documents as "bags of words"

- A *bag* or a *multiset* is an *unordered collection*
  (a *set* that can contain *more than one instance* of each element
- "Documents are 'bags of words' " means *word order is ignored*
- A "bag of words" retrieval system treats the following documents identically:
  - *man bites dog*
  - *dog bites man*
  - *dog man bites*
- "Bags of words" models can be surprisingly good

MACQUARIE UNIVERSITY

Boolean retrieval

- The Boolean model is arguably the simplest model to base an information retrieval system on.
- Queries are Boolean expressions, e.g., *Caesar* AND *Brutus*
- The seach engine returns all documents that satisfy the Boolean expression.

Does Google use the Boolean model?

Does Google use the Boolean model?

- On Google, the default interpretation of a query $[w_1\ w_2\ \ldots w_n]$ is w_1 AND w_2 AND ... AND w_n
- Cases where you get hits that do not contain one of the w_i:
  - anchor text
  - page contains variant of w_i (morphology, spelling correction, synonym)
  - long queries (n large)
  - boolean expression generates very few hits
- Simple Boolean vs. Ranking of result set
  - Simple Boolean retrieval returns matching documents in no particular order.
  - Google (and most well designed Boolean engines) *rank* the result set – they rank good hits (according to some estimator of relevance) higher than bad hits.

Boolean queries

- The Boolean retrieval model can answer any query that is a Boolean expression.
  - Boolean queries are queries that use AND, OR and NOT to join query terms.
  - Views each document as a set of terms.
  - Is precise: Document matches condition or not.
- Primary commercial retrieval tool for 3 decades
- Many professional searchers (e.g., lawyers) still like Boolean queries.
  - You know exactly what you are getting.

Unstructured data in 1650

- Which plays of Shakespeare contain the words *Brutus* AND *Caesar* AND NOT *Calpurnia*?
- *grep* (search) through all of Shakespeare's plays for *Brutus* and *Caesar*, then remove plays containing *Calpurnia*.
- Why is `grep` not the solution?
  - Slow (for large collections)
  - "NOT *Calpurnia*" is non-trivial
  - Ranked retrieval (find best document)
- Idea behind *indexing* for information retrieval
  - build an *inverted index* to speed retrieval
  - building the index is slow, but it only needs to be *built once*,
  - index can be built *off-line*, i.e., before queries have been seen

MACQUARIE UNIVERSITY

Term-document incidence matrix

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	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	
...							

- Entry is 1 (True) if term occurs in document.
  Example: *Calpurnia* occurs in *Julius Caesar*.
- Entry is 0 (False) if term doesn't occur in document.
  Example: *Calpurnia* doesn't occur in *The tempest*.

Retrieval using incidence vectors

- So we have a 0/1 vector for each term.
- To answer the query:
  Brutus AND *Caesar* AND NOT *Calpurnia*:
  - Take the vectors for *Brutus*, *Caesar*, and *Calpurnia*
  - Bitwise negate the vector of *Calpurnia*
    - NOT *Calpurnia* = NOT 010000 = 101111
  - Do a (bitwise) AND on the three vectors
  - 110100 AND 110111 AND 101111 = 100100

Boolean retrieval using incidence matrix for *Brutus* AND *Caesar* AND NOT *Calpurnia*

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	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	
...							
result:	1	0	0	1	0	0	

MACQUARIE UNIVERSITY

Answers to query

Anthony and Cleopatra, Act III, Scene ii

Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

Hamlet, Act III, Scene ii

Lord Polonius: I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Outline

MACQUARIE UNIVERSITY

Incidence matrix is impractical for big collections

- Consider a collection with:
  - $N = 10^6$ documents, each with about 1,000 *tokens*
  - $M = 500{,}000$ different terms

$\Rightarrow$ Incidence matrix has $10^6 \times 500{,}000 =$ *500 billion entries*

- But the matrix has no more than 1 billion 1s *(why?)*
  - extremely *sparse* (500$\times$0s for each 1)
  - use a representation that *only records the 1s*

Inverted Index

For each term t, store *a list of all documents that contain t.*

Dictionary		Postings								
Brutus	⟶	1	2	4	11	31	45	173	174	
Caesar	⟶	1	2	4	5	6	16	57	132	...
Calpurnia	⟶	2	31	54	101					
⋮										

dictionary **postings**

MACQUARIE UNIVERSITY

Document retrieval using an inverted index

- An *inverted index* maps *terms* to *the documents that contain them*
  - it "inverts" the collection (which maps documents to the words they contain)
  - will permit us to answer boolean queries without visiting entire corpus
- An inverted index is slow to construct (requires visiting entire corpus)
  - but this only needs to be *done once*
  - can be used for any number of queries
  - can be done before any queries have been seen
- Usually the *dictionary* is kept in RAM, but the *postings lists* (the documents for each term in dictionary) are stored on hard disk

Inverted index construction

1. Collect the documents to be indexed:
   | Friends, Romans, countrymen. | | So let it be with Caesar | . . .
2. Tokenize the text, turning each document into a list of tokens:
   | Friends | | Romans | | countrymen | | So | . . .
3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms: | friend | | roman | | countryman | | so | . . .
4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

Constructing an inverted index in Python

- Documents: NLTK corpora in *Gutenberg collection*
  - import nltk makes the collection available (if you've installed NLTK and the NLTK data)
  - nltk.corpus.gutenberg.fileids() returns a list of names of Gutenberg files

```
>>> import nltk
>>> nltk.corpus.gutenberg.fileids()
['austen-emma.txt', 'austen-persuasion.txt', ]
```

- Inverted index is a *dictionary* mapping each word *token* to a *set of file names*
  - gutenberg.words(filename) returns a list of words in filename

Constructing an inverted index in Python: the code

```
import nltk, collections

def make_inverted_index(corpus):
    inverted_index = collections.default_dict(set)
    for filename in corpus.fileids():
        for term in corpus.words(filename):
            inverted_index[term].add(filename)
    return inverted_index
```

Constructing an inverted index in Python: notes

```
def make_inverted_index(corpus):
    inverted_index = collections.default_dict(set)
    for filename in corpus.fileids():
        for term in corpus.words(filename):
            inverted_index[term].add(filename)
    return inverted_index
```

- The inverted index maps each term to a *set* of filenames
- If a term has not been seen before, default_dict *creates a set for it*

Outline

Duality: use set theory to do logic

- Instead of working with *Boolean vectors*, just use *sets containing the* True *elements*

Logical operation	**Set operation**
AND	intersection
OR	union
NOT	complement

Simple conjunctive query (two terms)

- Consider the query: *truth* AND *justice*
- To find all matching documents using inverted index:
    1. Locate *truth* in the dictionary
    2. Retrieve its postings list from the postings file
    3. Locate *justice* in the dictionary
    4. Retrieve its postings list from the postings file
    5. Intersect the two postings lists
    6. Return intersection to user

Simple conjunctive query in Python

```
def search1(inverted_index):
    truth_filenames = inverted_index["truth"]
    justice_filenames = inverted_index["justice"]
    return truth_filenames & justice_filenames
```

- & computes *set intersection*

More complex query in Python

```
def search2(inverted_index):
    brutus_filenames = inverted_index["Brutus"]
    caesar_filenames = inverted_index["Caesar"]
    calpurnia_filenames = inverted_index["Calpurnia"]
    return (brutus_filenames & caesar_filenames) - calpurnia_filenames
```

- − computes *set difference*

MACQUARIE UNIVERSITY

Running the searches in Python

```
>>> from wk02a import *
>>> inverted_index = make_inverted_index(nltk.corpus.gutenberg)
>>> search1(inverted_index)
set(['milton-paradise.txt', 'austen-emma.txt', 'chesterton-ball.txt', 'bible-k
>>> search2(inverted_index)
set(['shakespeare-caesar.txt', 'shakespeare-hamlet.txt'])
```

MACQUARIE UNIVERSITY

Query processing: Exercise

france ⟶ 1 → 2 → 3 → 4 → 5 → 7 → 8 → 9 → 11 → 12 → 13 → 14 → 15

paris ⟶ 2 → 6 → 10 → 12 → 14

lear ⟶ 12 → 15

Compute hit list for ((paris AND NOT france) OR lear)

Outline

Query optimisation

- Consider a query that is an AND of n terms, $n > 2$
- For each of the terms, get its postings list, then AND them together
- Example query: *Brutus* AND *Calpurnia* AND *Caesar*
- What is the best order for processing this query?

Query optimisation

- Example query: *Brutus* AND *Calpurnia* AND *Caesar*

Query optimisation

- Example query: *Brutus* AND *Calpurnia* AND *Caesar*
- Simple and effective optimisation: Process in order of increasing frequency

Query optimisation

- Example query: *Brutus* AND *Calpurnia* AND *Caesar*
- Simple and effective optimisation: Process in order of increasing frequency
- Start with the shortest postings list, then keep cutting further

Query optimisation

- Example query: *Brutus* AND *Calpurnia* AND *Caesar*
- Simple and effective optimisation: Process in order of increasing frequency
- Start with the shortest postings list, then keep cutting further

Brutus ⟶ 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174

Calpurnia ⟶ 2 → 31 → 54 → 101

Caesar ⟶ 5 → 31

MACQUARIE UNIVERSITY

Query optimisation

- Example query: *Brutus* AND *Calpurnia* AND *Caesar*
- Simple and effective optimisation: Process in order of increasing frequency
- Start with the shortest postings list, then keep cutting further
- In this example, first *Caesar*, then *Calpurnia*, then *Brutus*

Brutus →	1 →	2 →	4 →	11 →	31 →	45 →	173 →	174
Calpurnia →	2 →	31 →	54 →	101				
Caesar →	5 →	31						

More general optimisation

- Example query: (*madding* OR *crowd*) AND (*ignoble* OR *strife*)
- Get frequencies for all terms
- Estimate the size of each OR by the sum of its frequencies (conservative)
- Process in increasing order of OR sizes
- *How should negation be handled?*
  - Example query: (NOT *strife*) AND *crowd*

MACQUARIE UNIVERSITY

Outline

Identifying the most important words in a document

- Automatically identifying the *most important words* of a document is useful for:
  - identifying key-words of a document
  - summarisation and gisting
- Tf.Idf (Term Frequency times Inverse Document Frequency) is a very simple way of doing this
  - Tf.Idf is a bag-of-words approach (i.e., only uses word-document counts; ignores word order)
- There are many more sophisticated ways of identifying the most important words
  - more important words may come early in a document

Term Frequency

- Inspiration: very important words in a document should *appear very often* in that document
- $\mathrm{Tf}(d, w) =$ number of times term w appears in document d
- Unfortunately, the *highest frequency words often tell us little about a document*. *(Why?)*

MACQUARIE UNIVERSITY

Term Frequency example

D1 : computers process data quickly
D2 : data computers use data quickly
D3 : programs run quickly

Term t	**Document** d	**Term frequency** $\mathbf{Tf}(d, w)$
computers	D1	1
computers	D2	1
computers	D3	0
data	D1	1
data	D2	2
data	D3	0
quickly	D1	1
quickly	D2	1
quickly	D3	1
...	...	...

MACQUARIE UNIVERSITY

Term Frequency meets the Gutenberg Corpus

- *shakespeare-hamlet.txt*
  Highest term frequency words: [('the', 860), ('and', 606), ('of', 576), ('to', 576), ('I', 553), ('you', 479), ('a', 449), ('my', 435), ('in', 359), ('it', 354)]
- *bible-kjv.txt*
  Highest term frequency words: [('the', 62103), ('and', 38847), ('of', 34480), ('to', 13396), ('And', 12846), ('that', 12576), ('in', 12331), ('shall', 9760), ('he', 9665), ('unto', 8940)]
- *carroll-alice.txt*
  Highest term frequency words: [('the', 1527), ('and', 802), ('to', 725), ('a', 615), ('I', 543), ('it', 527), ('she', 509), ('of', 500), ('said', 456), ('Alice', 396)]

Document Frequency

- Inspiration: very important words *shouldn't be very common*
- Document frequency is the *number of documents this word appears in*
- $\mathrm{Df}(c, w) =$ number of documents in corpus c containing w
- Note: Important words should have a *low document frequency*

$\Rightarrow$ Rank by *inverse* document frequency $1/\mathrm{Df}(c, w)$

Document frequency example

D1 : computers process data quickly

D2 : data computers use data quickly

D3 : programs run quickly

Term t	**Document frequency** $\mathbf{Df}(c, w)$	$1/\mathbf{Df}(c, w)$
computers	2	0.5
process	1	1
data	2	0.5
quickly	3	0.33
use	1	1
...	...	...

MACQUARIE UNIVERSITY

Inverse Document Frequency meets Gutenberg

- *shakespeare-hamlet.txt*
  Lowest document frequency words: [('forgone', 1), ('vncharge', 1), ('cheefe', 1), ('Combate', 1), ('Hamlets', 1), ('gamboll', 1), ('Carters', 1), ('Marcellus', 1), ('Spectators', 1), ('Blasting', 1)]
- *bible-kjv.txt*
  Lowest document frequency words: [('Hashubah', 1), ('Doeg', 1), ('Jehoash', 1), ('respecteth', 1), ('deserveth', 1), ('Libnah', 1), ('Peniel', 1), ('Myra', 1), ('Jedidiah', 1), ('holpen', 1)]
- *carroll-alice.txt*
  Lowest document frequency words: [('NEAR', 1), ('BEG', 1), ('BEE', 1), ('CURTSEYING', 1), ('Game', 1), ('barrowful', 1), ('punching', 1), ('blacking', 1), ('rosetree', 1), ('Lory', 1)]

A first try at Tf.Idf (DON'T USE)

- Idea: Combine Tf and Df into a single formula
- We want its value to be big when:
  - Tf is big, and
  - Df is small
- First try at Tf.Idf
  (Term Frequency *times* Inverse Document Frequency)

$$\text{Tf.Idf}(c, d, w) = \frac{\text{Tf}(d, w)}{\text{Df}(c, w)}$$

MACQUARIE UNIVERSITY

First try Tf.Idf example (DON'T USE)

D1 : computers process data quickly

D2 : data computers use data quickly

D3 : programs run quickly

t	d	$\mathbf{Tf}(d, w)$	$\mathbf{Df}(c, w)$	$\mathbf{Tf}(d, w)/\mathbf{Df}(c, w)$
computers	D1	1	2	0.5
computers	D2	1	2	0.5
computers	D3	0	2	0
data	D1	1	2	0.5
data	D2	2	2	1
data	D3	0	2	0
quickly	D1	1	3	0.33
quickly	D2	1	3	0.33
quickly	D3	1	3	0.33
...	...	...	...	...

First try Tf.Idf meets Gutenberg

- *shakespeare-hamlet.txt*
  Highest Tf.Idf v0 words: [('Ham', 168.5), ('Qu', 62.0), ('Laer', 60.0), ('Ophe', 56.0), ('haue', 53.6), ('Pol', 49.0), ('the', 47), ('Hor', 47.5), ('Rosin', 43.0), ('Horatio', 40.0)]
- *bible-kjv.txt*
  Highest Tf.Idf v0 words: [('the', 3450), ('LORD', 2217.0), ('and', 2158), ('of', 1915), ('unto', 1490.0), ('to', 744), ('And', 713), ('that', 698), ('in', 685), ('saith', 631.0)]
- *carroll-alice.txt*
  Highest Tf.Idf v0 words: [('Alice', 132), ('the', 84), ('Mock', 56), ('Gryphon', 55), ('Hatter', 55), ('and', 44), ('Duchess', 42), ('to', 40), ('Dormouse', 40), ('a', 34)]

Tf.Idf as used in this class

- General intuition is that Tf.Idf version 0 gives rare words too high a score

⇒ Tweak formula to put less weight on document frequency
  - what about the *the*, *and*, *of*, etc., in the output?
  - use a *stop-list* containing 100 *most frequent words in corpus*
  - the new Tf.Idf formula will deal with these

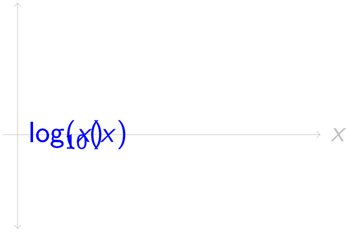
- *Tf.Idf formula used in this class:*

$$\mathbf{Tf.Idf}(c, d, w) = \mathbf{Tf}(d, w) \log \left(\frac{N}{\mathbf{Df}(c, w)} \right)$$

where $N =$ *number of documents in collection*

MACQUARIE UNIVERSITY

A brief reminder about logarithms



- Logarithms are calculated with respect to a *base*
  - I'm using logarithms base $e \approxeq 2.718$, a.k.a. *natural logarithms*, sometimes also written $\ln(x)$ or $\log_e(x)$
  - Logarithms base 10 are also common; these are written $\log_{10}(x)$
  - Logarithms with different bases only *differ by a scaling factor*, $\log_{10}(x) \approxeq 2.3 \times \log_e(x)$
- The logarithm of 1 is 0, or in maths $\log(1) = 0$
- Since we want the words or documents that *score highest under Tf.Idf*, it doesn't matter which base we use for our logarithms

MACQUARIE UNIVERSITY

Tf.Idf example

D1 : computers process data quickly

D2 : data computers use data quickly

D3 : programs run quickly

t	d	$\mathbf{Tf}(d, w)$	$\mathbf{Df}(c, w)$	$N/\mathbf{Df}(c, w)$	$\mathbf{Tf}(d, w)\log(N/\mathbf{Df}(c, w))$
computers	D1	1	2	1.5	0.40
computers	D2	1	2	1.5	0.40
computers	D3	0	2	1.5	0
data	D1	1	2	1.5	0.40
data	D2	2	2	1.5	0.80
data	D3	0	2	1.5	0
quickly	D1	1	3	1	0
quickly	D2	1	3	1	0
quickly	D3	1	3	1	0
...	...	...	...	...	...

Tf.Idf meets Gutenberg

- *shakespeare-hamlet.txt*
  Highest Tf.Idf words: [('Ham', 740), ('haue', 288), ('Hor', 208), ('Qu', 179), ('Hamlet', 177), ('Laer', 173), ('Ophe', 161), ('Pol', 141), ('Rosin', 124), ('selfe', 118)]
- *bible-kjv.txt*
  Highest Tf.Idf words: [('LORD', 11916), ('unto', 9821), ('Israel', 2827), ('saith', 2772), ('David', 1906), ('Judah', 1792), ('hath', 1551), ('shalt', 1118), ('Jesus', 1073), ('thereof', 995)]
- *carroll-alice.txt*
  Highest Tf.Idf words: [('Alice', 709), ('Mock', 161), ('Gryphon', 158), ('Hatter', 158), ('Turtle', 129), ('Duchess', 121), ('Dormouse', 115), ('Rabbit', 80), ('Caterpillar', 78), ('Hare', 55)]

Outline

MACQUARIE UNIVERSITY

Using Tf.Idf to rank search results

- Inspiration: query terms should be important terms in document
  - use Tf.Idf to measure how important each query term is
  - rank documents by the *sum of their Tf.Idf scores for query words*
- Problem: long documents have higher Tf.Idf scores
- Solution: *scale* the Tf.Idf scores by dividing by document length

$$\text{Score}(c, d, ts) = \frac{1}{|d|} \sum_{t \in ts} \text{Tf.Idf}(c, d, t)$$

where *ts are the search terms* and $|d|$ *is the length of document* d.

MACQUARIE UNIVERSITY

Scaled Tf.Idf retrieval example

D1 : computers process data quickly

D2 : data computers use data quickly

D3 : programs run quickly

Query: data computers

- Conjunctive Boolean query returns **D1** and **D2**

t	d	$\textbf{Tf.Idf}(c, d, w)$	$\textbf{Tf.Idf}(c, d, w)/\lvert d \rvert$
computers	D1	0.40	0.10
computers	D2	0.40	0.08
computers	D3	0	0
data	D1	0.40	0.10
data	D2	0.80	0.16
data	D3	0	0

- Score(c, **D1**, *data* AND *computers*) $= 0.20$
  Score(c, **D2**, *data* AND *computers*) $= 0.24$
- So ranked retrieval results are **D2**, **D1**

Outline

Relevance feedback

- Idea: Use user feedback to improve document ranking
  - Users inspect documents in some order
  - After the user has inspected a document, they can tell you if it's relevant
  - Use the user-supplied relevance information about current document to rank the remaining documents
- Example:
  - User has identified a set R of relevant documents
  - Use e.g., Tf.Idf to find *most important words* W in R
  - Conduct a ranked search for W, and return results to user

Query expansion

- Queries are often *missing relevant terms*
  - ⇒ low recall (relevant documents are not retrieved)
- *Query expansion* adds related words to query
- Example:
  - User query: *cheap* AND *car*
  - Expanded query: *(cheap* OR *inexpensive)* AND *(car* OR *automobile)*
- Standard way to perform query expansion is using a *thesaurus*, which lists *synonyms* for words

MACQUARIE UNIVERSITY

Query expansion via Pseudo-relevance feedback

- Idea: *Use search results to find new relevant search terms*
  1. Search for user's original query, returning documents R_0
  2. Identify key words W in R_0 (e.g., with modified Tf.Idf)
  3. Run a new approximate search for W, returning documents R_1
  4. Rank $R_0 \cup R_1$ and return to user
- This works because synonyms often appear in the same document

Outline

Information Retrieval

Inverted index

Processing Boolean queries with an inverted index

Query optimisation

Term Frequency and Inverse Document Frequency

Using Tf.Idf to rank search results

More sophisticated retrieval techniques

Summary

MACQUARIE UNIVERSITY

Review of Boolean information retrieval

- *Bag of words* assumption: *ignore word order*
- Boolean retrieval defines relevant documents using *Boolean operations on term-document incidence matrix*
- Making search practical on large collections:
  - searching by inspecting all documents (grep-search) is impractically slow
  - term-document incidence matrix is too big
  - *inverted index* is a practical solution

Document retrieval using an inverted index

- An *inverted index* maps each *term* to *the documents that contain it*
  - it “inverts” the collection (which maps documents to the words they contain)
  - will permit us to answer boolean queries without visiting entire corpus
- An inverted index is slow to construct (requires visiting entire corpus)
  - but this only needs to be *done once*
  - can be used for any number of queries
  - can be done before any queries have been seen
- Usually the *dictionary* is kept in RAM, but the *postings lists* (the documents for each term in dictionary) are stored on hard disk

Ranking search results and query expansion

- Tf.Idf and similiar methods can *identify the most important terms in a document*
- This can be used to *rank search results* by how well the query terms match the important words in the document
- *Query expansion* often improves recall in information retrieval by retrieving documents with words not appearing the query