

HES-SO - University of Applied Sciences of western Switzerland - MSE

Introduction to Information Retrieval

Resume of the MSE lecture

by

Jérôme KEHRLI

Largeley inspired from

"Introduction to information retrieval" C.D Manning et al. / Cambridge University Press,
2008

"Search Engines, Information Retrieval in Practice", W.B Croft, D. Metzler, T. Strohman,
Persone Education, 2010

prepared at HES-SO - Master - Provence,
written Oct-Dec, 2010

Resume of the Data management lecture

Abstract:

TODO

Keywords: Data management, Data mining, Market Basket Analysis

Contents

1	Introduction	1
1.1	Search and Information retrieval	1
1.1.1	The notion of <i>Document</i>	2
1.1.2	Comparing text	2
1.1.3	Dimensions of IR	3
1.1.4	Big issues in IR	3
1.1.5	Three scales for IR systems	4
1.2	IR System architecture	4
1.2.1	Indexing process	5
1.2.2	Query process	8
1.3	IR Models	9
1.3.1	Ad-hoc Retrieval	10
1.3.2	Query relevance	10
1.4	Practice	10
1.4.1	Stopping and Stemming	10
2	Boolean retrieval model	13
2.1	Principle	13
2.2	Term-document incidence matrix	14
2.2.1	Introductory example	14
2.2.2	Term-document incidence matrix	14
2.2.3	Answers to query	15
2.2.4	The problem : bigger collections	15
2.3	Inverted index	15
2.3.1	Token sequence	16
2.3.2	Sort postings	16
2.3.3	Dictionary and Postings	17
2.3.4	Query processing	17
2.4	Boolean queries	20

2.4.1	Example: Westlaw	20
2.5	Query Optimization	20
2.5.1	Process in increasing frequency order	21
2.5.2	More general optimization	21
2.6	Practice	21
2.6.1	Inverted Index	21
2.6.2	Boolean queries	22
3	Scoring, term weighting and the vector space model	25
3.1	The Problem with Boolean search	26
3.2	Feast of famine	26
3.3	Ranked retrieval	26
3.3.1	Scoring	26
3.3.2	1st naive attempt: The Jaccard coefficient	27
3.3.3	Term frequency	27
3.3.4	Document frequency	29
3.3.5	Effect of idf on ranking	30
3.4	tf-idf weighting	30
3.4.1	1st naive query implementation : simply use tf-idf for ranking documents . .	31
3.4.2	Weight matrix	31
3.4.3	Consider documents as vectors	31
3.4.4	Consider queries as vectors as well	32
3.4.5	Formalize vector space similarity	32
3.5	Use angle to rank document	33
3.5.1	From angle to cosine	33
3.5.2	length normalization	33
3.5.3	Compare query and documents - rank documents	34
3.5.4	Cosine example	35
3.6	General tf-idf weighting	37
3.6.1	Components of tf-idf weighting	37
3.6.2	Computing cosine scores	38
3.6.3	Conclusions	39

3.7	Practice	40
3.7.1	idf and stop words	40
3.7.2	idf logarithm base	40
3.7.3	Euclidean distance and cosine	41
3.7.4	Vector space similarity	42
3.7.5	Request and document similarities	42
3.7.6	Various questions	43
4	Evaluation	45
4.1	Introduction	45
4.1.1	Evaluation corpus	46
4.2	Effectiveness Measures	46
4.2.1	Recall	47
4.2.2	Precision	47
4.2.3	Trade-off between Recall and Precision	47
4.2.4	Classification errors	48
4.2.5	F Measure	48
4.3	Ranking Effectiveness	49
4.3.1	Summarizing a ranking	49
4.3.2	AP - Average precision	50
4.3.3	MAP - Mean Average Precision	50
4.3.4	Recall-Precision graph	51
4.3.5	Interpolation	52
4.3.6	Average Precision at Standard Recall Levels	53
4.4	Focusing on top documents	54
4.5	Practice	54
4.5.1	Precision and Recall	54
4.5.2	Search systems comparison	55
4.5.3	Search system analysis	56
4.5.4	Search systems comparison (another example)	58
4.5.5	F-measure	59

5	Queries and Interfaces	61
5.1	Introduction	61
5.1.1	Information Needs	61
5.1.2	Queries and Information Needs	62
5.1.3	Interaction	62
5.1.4	ASK Hypothesis	62
5.2	Query-based Stemming	62
5.2.1	Stem Classes	62
5.3	Spell checking	63
5.3.1	Basic Approach	63
5.3.2	Noisy channel model	64
5.4	Relevance feedback	64
5.4.1	Query reformulation	64
5.4.2	Optimal query	65
5.4.3	Standard Rocchio Method	65
5.5	practice	66
5.5.1	Relevance feedback	66

Introduction

Contents

1.1 Search and Information retrieval	1
1.1.1 The notion of <i>Document</i>	2
1.1.2 Comparing text	2
1.1.3 Dimensions of IR	3
1.1.4 Big issues in IR	3
1.1.5 Three scales for IR systems	4
1.2 IR System architecture	4
1.2.1 Indexing process	5
1.2.2 Query process	8
1.3 IR Models	9
1.3.1 Ad-hoc Retrieval	10
1.3.2 Query relevance	10
1.4 Practice	10
1.4.1 Stopping and Stemming	10

1.1 Search and Information retrieval

Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within a large collection (usually stored on computers).

Why is this an important matter ?

- Search on the web is a daily activity for many people throughout the world
- Search and communication are most popular uses of a computer
- Applications involving search are everywhere
- The field of computer science that is most involved with R&D for search is *Information Retrieval*

"Information Retrieval is a field concerned with the structure, analysis, organisation, storage, searching and retrieval of information" - Salton, 1968

This general definition can be applied to many types of information and search applications. The primary focus of *IR* since the 50s has mostly been on *text* and *documents*.

1.1.1 The notion of *Document*

1.1.1.1 What is a document ?

Examples: web pages, emails, books, news stories, scholarly papers, text messages, Ms Office documents, PDFs, forum postings, patents, IM sessions, etc.

Common properties :

- Significant text content
- Some structure (e.g. title, author, date for papers, subject, sender and destination for emails, etc.)

1.1.1.2 Documents vs. Database records

Database records

- SQL is very structured
- Database records (or *tuples* in a relational database) are typically made up of well-defined fields (or attributes), e.g. bank records with account numbers, balances, names, addresses, security numbers, dates, etc.)
- It is easy to compare fields with well-defined semantics to queries in order to find matches.

Example for a bank database query :

- *Find records with balance > \$50'000 in branches located in Lausanne, Switzerland*
- Matches are easily found by comparison with fields values of records.

Documents

- The query is not that much structured
- Text is in a general manner much more complicated

Example for a search engine query :

- *query : bank scandals in Switzerland*
- This text must be compared to the text of the entire news story

1.1.2 Comparing text

Comparing the query text to the document text and determining what is a good match is the *core issue* of information retrieval.

Exact matching of words is not enough :

- Many different ways to write the same thing in a "natural language" like english.
- E.g. does a news story containing the text "*bank director in Zürich sets funds*" match the previous example query ?
- Some stories will be better matches than others

1.1.3 Dimensions of IR

- IR is more than just text and more than just web search - although these are central.
- People doing IR work with different **content** (or media), different **types of search applications** and different **tasks**

Content	Applications	Tasks
Text	Web search	Ad hoc search * ₃
Images * ₁	Vertical Search * ₂	Filtering * ₄
Video * ₁	Enterprise Search	Classification * ₅
Scanned docs	Desktop search	Question answering
Audio	Forum Search	
Music	P2P Search	
	Literature Search	

*₁ Search of forms, patterns, logos

*₂ This is a full domain on its own

*₃ For instance on google.com, at each new request

*₄ This is a special kind where the request is fixed but the documents flow varies

*₅ For creating portals like yahoo category, etc.

1.1.3.1 IR tasks

Ad-hoc search : Find relevant documents for an arbitrary text query

Filtering : Identify relevant user profiles for a new document

Classification : Identify relevant labels for a document

Question answering : Give a specific answer to a question. This usually involves natural language analysis such as on Wolfram Alpha.

1.1.4 Big issues in IR

1.1.4.1 Relevance

What is relevance ?

Simple definition : A relevant document contains the information that a person was looking for when she submitted a query to the search engine.

A document is relevant for a specific query if it answers the question of the user.

Many factors influence a person's decision about what is relevant, e.g. task, context, novelty, etc. One can distinguish *topical relevance* (same topic) versus *user relevance* (everything else).

- *Retrieval models* define a view on relevance
- *Ranking algorithms* used in search engine are based on *Retrieval models*.
- Most models describe statistical properties of text rather than linguistic. i.e. counting simple text features such as words instead of parsing and analyzing the sentences.

1.1.4.2 Evaluation

The **Evaluation** of a *Retrieval model* is about experimental procedures and measures for comparing system output with user expectations.

- IR evaluation methods are now used in many fields
- Typically use *test collections* of documents, queries and relevance judgments **most commonly used/known as TREC collections**.
- ***Recall* and *Precision* are two examples of *effectiveness* (efficacité) measures.**

1.1.4.3 User and Information Needs

- Search evaluation is user-centered.
- Keyword queries are often poor descriptions of actual information needs.
- Interaction and context are often important for understanding user intent

Hence the needs for query refinement techniques such as *query expansion*, *query suggestion*, *relevance feedback* improve ranking.

1.1.5 Three scales for IR systems

- **Web Search** : Search over billions of documents on Internet (e.g. google)
- **Personnal information retrieval** : Broad range of document but limited number (e.g. Max OSX spotlight, search integrated in the OS)
- **Enterprise, Institutional or domain-specific search** : Search in collections such as corporation's internal documents, etc.

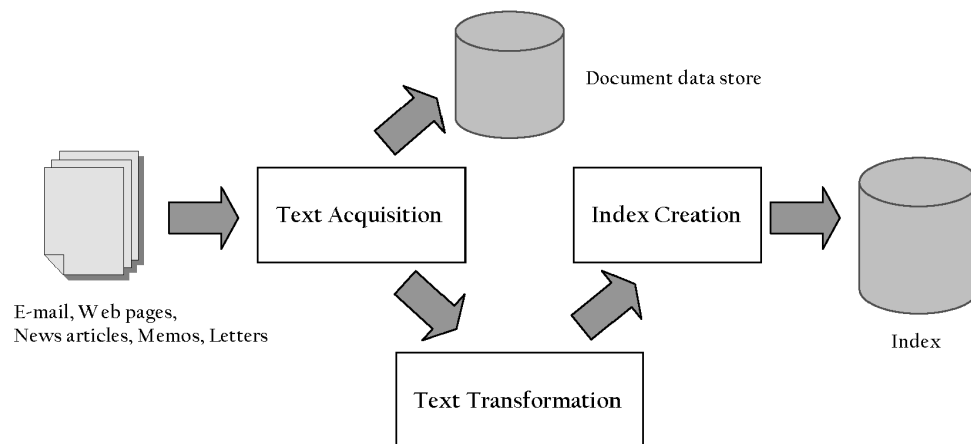
1.2 IR System architecture

A software architecture consists of software components, the interfaces provided by those components, and the relationships between them. Architecture describes a system at a particular level of abstraction.

Architecture of a search engine is determined by two requirements :

- **Effectiveness** (quality of results)
- **Efficiency** (response time and throughput)

1.2.1 Indexing process


Text acquisition

identifies and stores document for indexing

Text transformation

transforms documents into index terms or features

Index creation

takes index terms and creates data structures (indexes) to support fast searching.

1.2.1.1 Text Acquisition

→ **The crawler:**

- Identifies and acquires documents for search engine
- Many types : web, enterprise, desktop
- Web crawlers follow links to find documents
 - Must efficiently find huge numbers of web pages (coverage) and keep them up-to-date (freshness)
 - Single site crawlers for site search
 - Topical or focused crawlers for vertical search
- Document crawlers for enterprise and desktop search (Follow links and scan directories)

→ **Feeds:**

- Real-time streams of documents, e.g., web feeds for news, blogs, video, radio, tv
- RSS is common standard. RSS "reader" can provide new XML documents to search engine

→ **Conversion:**

- Convert variety of documents into a consistent text plus metadata format, e.g. HTML, XML, Word, PDF, etc. → XML
- Convert text encoding for different languages. e.g. using a Unicode standard like UTF-8

→ **Document data store:**

- Stores text, metadata, and other related content for documents
 - Metadata is information about document such as type and creation date
 - Other content includes links, anchor text
- Provides fast access to document contents for search engine components, e.g. result list generation.
- Could use relational database system, e.g. for a small indexing system ...
- ... but more typically, a simpler, more efficient storage system is used due to huge numbers of documents (.e.g google's *BigTable*)

1.2.1.2 Text transformation

→ **Parser:**

- Processing the sequence of text tokens in the document to recognize structural elements, e.g., titles, links, headings, etc.
- Tokenizer recognizes "words" in the text. It must consider issues like capitalization, hyphens, apostrophes, non-alpha characters, separators
- Markup languages such as HTML, XML often used to specify structure
 - Tags used to specify document elements, e.g., <h2> Overview </h2>
 - Document parser uses syntax of markup language (or other formatting) to identify structure

→ **Stopping:**

- Remove common words, e.g., "and", "or", "the", "in"
- **these words are called stop words**
- Some impact on efficiency and effectiveness
- Can be a problem for some queries

→ **Stemming:**

- Group words derived from a common *stem* (or *lem*, in french "*Lemmatisation*"), e.g., "computer", "computers", "computing", "compute"
- Usually effective, but not for all queries
- Benefits vary for different languages

→ **Link analysis:**

- Makes use of links and anchor text in web pages
- Link analysis identifies popularity and community information, e.g., PageRank
- Anchor text can significantly enhance the representation of pages pointed to by links
- Significant impact on web search, less importance in other applications

→ **Information Extraction:**

- Identify classes of index terms that are important for some applications, e.g., *named entity recognizers* identify classes such as people, locations, companies, dates, etc.

→ **Classifier:**

- Identifies class-related metadata for documents
 - i.e., assigns labels to documents
 - e.g., topics, reading levels, sentiment, genre
- Use depends on application

1.2.1.3 Index creation

→ **Document statistics:**

- Gathers counts and positions of words and other features
- Used in ranking algorithm

→ **Weighting:**

- Computes weights for index terms
- Used in ranking algorithm
- e.g., **tf.idf weight** : Combination of term frequency in document and inverse document frequency in the collection

→ **Inversion:**

- Core of indexing process
- Converts document-term information to term-document for indexing: difficult for very large numbers of documents
- Format of inverted file is designed for fast query processing
 - Must also handle updates
 - Compression used for efficiency

→ **Index distribution:**

- Distributes indexes across multiple computers and/or multiple sites
- Essential for fast query processing with large numbers of documents
- Many variations : Document distribution, term distribution, replication
- P2P and *distributed IR* involve search across multiple sites

- Highlights important words and passages
- Retrieves appropriate advertising in many applications
- May provide clustering and other visualization tools

1.2.2.2 Ranking

→ **Scoring:**

- Calculates scores for documents using a ranking algorithm
- Core component of search engine
- Basic form of score is $\sum q_i d_i$
 - q_i and d_i are query and document term weights for term i
- Many variations of ranking algorithms and retrieval models

→ **Performance optimization:**

Designing ranking algorithms for efficient processing :

- Term-at-a-time vs. document-at-a-time processing
- Safe vs. unsafe optimizations

→ **Distribution:**

- Processing queries in a distributed environment
- Query broker distributes queries and assembles results
- Caching is a form of distributed searching

1.2.2.3 Evaluation

The whole purpose of evaluation is to help make the next queries better solved.

→ **Logging:**

- Logging user queries and interaction is crucial for improving search effectiveness and efficiency
- Query logs and click through data used for query suggestion, spell checking, query caching, ranking, advertising search, and other components

→ **Ranking analysis:** Measuring and tuning ranking effectiveness

→ **Performance analysis:** Measuring and tuning system efficiency

1.3 IR Models

TODO Put first image of page 21

1.3.1 Ad-hoc Retrieval

- Our goal : to develop a system to address the ad hoc retrieval task
- Ad hoc retrieval : the most standard IR task. In this context, a system aims to provide documents from within the collection that are relevant to [an arbitrary user information need](#), communicated to the system by means of a [one-off, user-initiated query](#)

The user expresses a need of information through a request → this request needs to be represented under a boolean, algebraic, etc. form, e.g. "Java AND Hibernate AND Spring".

1.3.2 Query relevance

- An [information need](#) is the topic about which the user desires to know more, and is differentiated from a [query](#), which is what the user conveys to the computer in an attempt to communicate this information need.
- A document is [relevant](#) if it is one that the user perceives as containing information of value with respect to their personal information need.
- A document is [relevant](#) if it matches the information need of the user (→ this can be hard to define)

1.4 Practice

1.4.1 Stopping and Stemming

Stopping consists in removing most common words with no relevance at all (such as "and", "or", etc.). Stemming consists in grouping words on a common stem. Both are text transformation operations applied before indexing.

1.4.1.1 Advantages

What are the advantages of both techniques related to the search ?

Stopping:

- Reduction of the index size
- The precision is potentially increased
- Gain in terms of performance (less terms to consider)

Stemming:

- The dictionary is reduced: gain in terms of storage space and performance
- The scope is enlarged and we get a search that is more tolerant in language variations

1.4.1.2 Drawbacks

What are the drawback of both techniques related to the search ?

Stopping:

- Potential precision loss : what if someone searches for the exact movie title "Harry and Sally" ?

Stemming:

- Lost of information: potential precision loss.

Boolean retrieval model

Contents

2.1 Principle	13
2.2 Term-document incidence matrix	14
2.2.1 Introductory example	14
2.2.2 Term-document incidence matrix	14
2.2.3 Answers to query	15
2.2.4 The problem : bigger collections	15
2.3 Inverted index	15
2.3.1 Token sequence	16
2.3.2 Sort postings	16
2.3.3 Dictionary and Postings	17
2.3.4 Query processing	17
2.4 Boolean queries	20
2.4.1 Example: Westlaw	20
2.5 Query Optimization	20
2.5.1 Process in increasing frequency order	21
2.5.2 More general optimization	21
2.6 Practice	21
2.6.1 Inverted Index	21
2.6.2 Boolean queries	22

2.1 Principle

The boolean retrieval model is a model for information retrieval in which we can pose any [query](#) which is in the form of a [Boolean expression of terms](#)

- Boolean expression: terms are combined with operators **AND**, **OR** and **NOT**
- Example of a query : "Ceasar AND Brutus"

The search engines returns all documents that satisfy the Boolean expression. Obviously this model is very limited as it is not able to return the document that matches only partially the request not is it able to perform any ranking, i.e search engines such as google use a different model.

2.2 Term-document incidence matrix

2.2.1 Introductory example

Question : which plays of Shakespeare contain the words **Brutus** AND **Caesar** but NOT **Calpurnia** ?

- One could grep all of Shakespeare's plays for **Brutus** and **Caesar**, then strip out lines containing **Calpurnia**?
- Why is that not the solution ?
 - Slow (for large corpora)
 - NOT Calpurnia is non-trivial
 - Other operations (e.g., find the word Romans near countrymen) not feasible
 - Ranked retrieval (best documents to return)

2.2.2 Term-document incidence matrix

The way to avoid linearly scanning the texts for each query is to **index** the documents in advance. The following is a *binary representation of the index*, the term-document incidence matrix:

	Documents →					
	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

↓ Terms

Document(s) vector(s)

1 if play contains word, 0 otherwise

- Entry is 1 if terms occurs. Example : Calpurina occurs in Julius Ceasar.
- Entry is 0 if terms does not occurs. Example : Calpurina doesn't occur in The Tempest.
- We record for each document whether it contains each word out of all the words Shakespeare used.
- Terms are the indexed units : they are usually words but may also be names, symbols, etc.

Now dependening on whether we look at the matrix rows or columns, we can have:

- a **vector for each term**, which shows the document it appears in, or
- a **vector for each document**, showing the terms that occur in it.

2.2.3 Answers to query

To answer the query **Brutus AND Ceasar AND NOT Calpurnia** :

- Take the vectors for **Brutus**, **Ceasar** and the complement of the vector for **Calpurnia**
- $110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$
- This the answer to this query are "Anthony and Cleopatra" and "Hamlet"

2.2.4 The problem : bigger collections

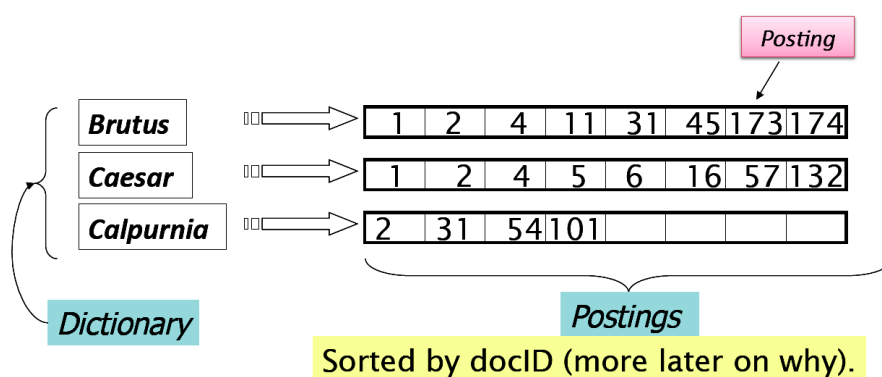
- Consider $N = 10^6$ documents, each with about 1000 words
- On average 6 bytes per token $\Rightarrow$ size of document collection is about 6Gb.
- Assume there is $M = 500'000$ distinct terms in the collection $\Rightarrow M = 500'000 \times 10^6 =$ half a trillion 0s and 1s.
- BUT the matrix has no more than a billion 1s $\Rightarrow$ Matrix is extremely sparse (few 1s for all the 0s)

We have to find a better representation for the matrix. The idea is to store only the 1s. This is what we do with inverted indexes.

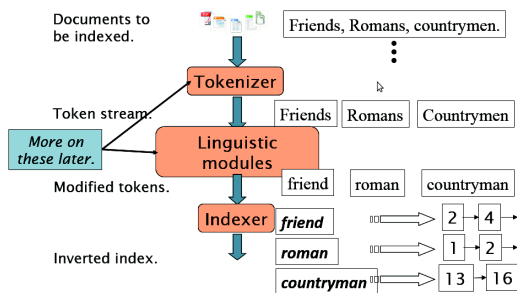
2.3 Inverted index

For each term t , we must store a list of all documents that contain $t \rightarrow$ identified each by a **docID** = document serial number.

This **docID** is typically assigned during the index construction by giving successive integers to each new document that is encountered.



Inverted indexes 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:

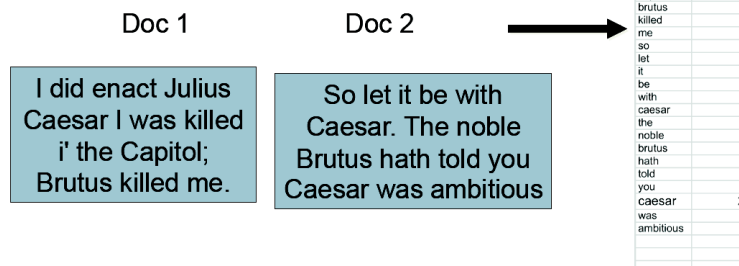
Friends Romans countrymen So ...

4. **Index** the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

This is considered an inversed index because the term gives the document list, and not the other way around. *The term inverted is actually redundant as an index always maps back from terms to the parts of a document where they occur.*

2.3.1 Token sequence

- Sequence of (Modified token, Document ID) pairs.



2.3.2 Sort postings

The postings are lexically sorted.

Sort by terms

- And then docID

Core indexing step

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

2.3.3 Dictionary and Postings

First a bit of a terminology:

- We use the term **dictionary** for the whole data structure.
- We use the term **vocabulary** for the set of terms.

Create posting lists, determine document frequency:

Structure of storage:

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

Why frequency?
Will discuss later.

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

Terms
and
counts

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

Lists of
docIDs

Pointers

2.3.4 Query processing

How do we process a query with the index we just built ?

2.3.4.1 Simple conjunctive query - support example

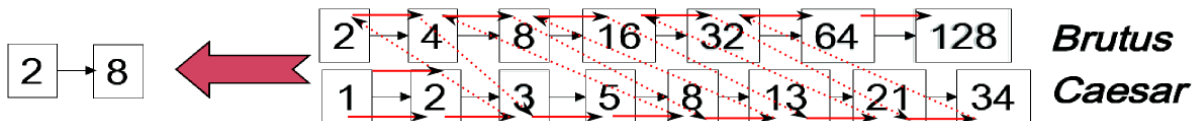
We will consider here the following query: **Brutus AND Calpurnia**. In order to find all matching documents using inverted index, one needs to:

1. Locate **Brutus** in the Dictionary;
2. Retrieve its postings.
3. Locate **Caesar** in the Dictionary;
4. Retrieve its postings.
5. "Merge" - Intersect - the two postings:
6. Return intersection to user

The merge - intersecting two postings list :

Principle:

- **Crucial : the lists need to be sorted**
- Walk through the two postings list with 2 pointers. Always move the pointer on the littlest element.
- When both pointer point at the same value, return the value in the intersecting list result.



- The intersection calculation occurs in time linear to the total number of posting entries
- If the list length are x and y , the merge takes $O(x + y)$ operations

2.3.4.2 INTERSECTION - Intersecting two postings lists

```

INTERSECT(p1, p2)
1  answer ← <>
2  while p1 ≠ NIL and p2 ≠ NIL
3  do if docID(p1) = docID(p2)
4      then ADD(answer, docID(p1))
5          p1 ← next(p1)
6          p2 ← next(p2)
7  else if docID(p1) < docID(p2)
8      then p1 ← next(p1)
9      else p2 ← next(p2)
10 return answer

```

One should note that if both lists are very similar, the algorithm behaves quite fast as both list are walked through simultaneously (see 5-6).

2.3.4.3 UNION - Unifying two postings lists

(Difference with intersection are underlined in blue.

```

UNION(p1, p2)
1  answer ← <>
2  while p1 ≠ NIL and p2 ≠ NIL
3  do if docID(p1) = docID(p2)
4      then ADD(answer, docID(p1))
5          p1 ← next(p1)
6          p2 ← next(p2)
7  else if docID(p1) < docID(p2)
8      then
9          ADD(answer, docID(p1))
10         p1 ← next(p1)
11     else
12         ADD(answer, docID(p2))
13         p2 ← next(p2)
14  # Now the problem is one list could be
15  # shorter than the other one
16  while p1 ≠ NIL ; do
17      ADD(answer, docID(p1))
18      p1 ← next(p1)
19  while p2 ≠ NIL ; do
20      ADD(answer, docID(p2))
21      p2 ← next(p2)
22  return answer

```

The principle here is always to add the values which are in both lists (4-7) and the littlest value as well it is guaranteed not to be in the other list.

Between 14-21 we take into consideration cases where one list is shorter than the other one quite easily. Simply both list are analyzed for remaining stuff which is added to the unified results.

2.3.4.4 INTERSECTION with negation - AND NOT

A naive implementation of the query y AND (NOT y) would be to evaluate (NOT y) first as a new postings list, which takes $O(N)$ time, and the merge it with x . Therefore, the overall complexity will be $O(N)$.

An efficient postings merge algorithm to evaluate x AND (NOT y) is:

(Difference with usual INTERSECTION is shown in red)

```

INTERSECT_NOT_SECOND (p1, p2)
1  answer ← <>
2  while p1 ≠ NIL and p2 ≠ NIL
3  do if docID(p1) = docID(p2)
4      then ADD(answer, docID(p1))
5          p1 ← next(p1)

```

```
6         p2 ← next(p2)
7     else if docID(p1) < docID(p2)
8         then ADD(answer, docID(p1))
9         p1 ← next(p1)
10        else p2 ← next(p2)
11 return answer
```

Same principle as UNION: we know that if the current p1 docID is the smallest, we won't find it in the other list.

2.4 Boolean queries

→ Exact match !

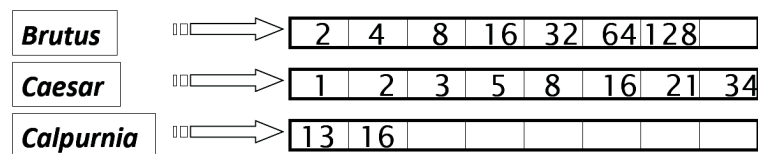
- The Boolean retrieval model is being able to ask a query that is a Boolean expression:
 - Boolean Queries are queries using AND, OR and NOT to join query terms
 - Views each document as a set of words
 - Is precise: document matches condition or not.
 - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades.
- Many professional searchers still like the Boolean model: you know exactly what you are(will be) getting.
- Many search systems you still use are Boolean: Email, library catalog, Mac OS X Spotlight

2.4.1 Example: Westlaw

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- Tens of terabytes of data; 700,000 users
- Majority of users still use boolean queries

2.5 Query Optimization

- What is the best order for query processing?
- Consider a query that is an *AND* of *n* terms.
- For each of the *n* terms, get its postings, then *AND* them together.



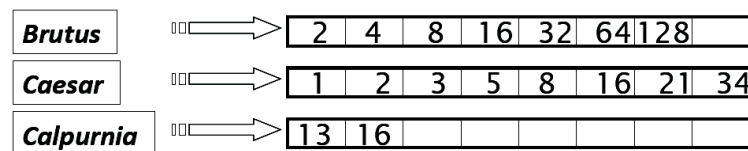
Query: *Brutus AND Calpurnia AND Caesar*

2.5.1 Process in increasing frequency order

Simple and efficient idea : process in order of increasing frequency - start with the smallest lists.

Start with the shortest postings list, then keep cutting further.

This requires one to keep the **Document frequency** for all terms. In this implementation, that simply is the size of the lists.



Execute the query as **(Calpurnia AND Brutus) AND Caesar**

In this example, we process first **Caesar**, then **Calpurnia**, then **Brutus**.

One should not think that this optimisation doesn't necessarily guarantee that the processing order will be optimal. There are simple counter-examples. For instance one can imagine a case where the two biggest amongst dozens of lists have no intersection at all, the optimal order in this case would be to process the two biggest first and as they have no intersection the whole processing would be over.

2.5.2 More general optimization

- The goal is to be able to optimize more complicated queries, e.g., (**madding** OR **crowd**) AND (**ignoble** OR **strife**).
- Get *Document frequencies* for all terms.
- **Estimate the size of each OR by the sum of its doc. freq.'s (conservative).**
- **Process in increasing order of OR sizes.**

2.6 Practice

2.6.1 Inverted Index

Draw the inverted index that would be built for the following document collection:

Doc 1 new home sales top forecast
Doc 2 home sales rise in july
Doc 3 increase in home saley in july
Doc 4 july new home sales rise

Result :

forecast	→	(1)	1				
home	→	(4)	1	2	3	4	
in	→	(2)	2	3			
increase	→	(1)	3				
july	→	(3)	2	3	4		
new	→	(2)	1	4			
rise	→	(2)	2	4			
sales	→	(4)	1	2	3	4	
top	→	(1)	1				

2.6.2 Boolean queries

Let the following be a set of documents:

D1: SDK, Android, Google, Mobile, Software
D2: Song, Android, Radiohead, Paranoid, Yorke
D3: SDK, System, Android, Kernel, Linux
D4: Android, Mobile, Google, Software, System
D5: Mobile, Swisscom, SMS, subscription, rate

And the following be a set of boolean queries:

R1: Android OR SDK OR Google OR Mobile
R2: Android AND SDK AND Google AND Mobile

According to the boolean model, what are the documents retrieved for each query ?

R1: **D1 D2 D3 D4 D5**
R2: **D1**

Formulize a criticism of both *AND* and *OR* operators

- *AND* is too restrictive
- *OR* is too permissive

the below assumes the reader is familiar with the concepts introduced in chapter 4

Let's assume D1, D3 and D4 are relevant. Compute recall and precision for these results computed above.

$$R1 = \text{RNRRN}$$

$$\text{Recall} = \frac{|A \cap B|}{|A|} = \frac{3}{3} = 1$$

$$\text{Precision} = \frac{|A \cap B|}{|B|} = \frac{3}{5} = 0.6$$

$$R2 = \text{R}$$

$$\text{Recall} = \frac{|A \cap B|}{|A|} = \frac{1}{3} = 0.33$$

$$\text{Precision} = \frac{|A \cap B|}{|B|} = \frac{1}{1} = 1$$

Scoring, term weighting and the vector space model

Contents

3.1 The Problem with Boolean search	26
3.2 Feast of famine	26
3.3 Ranked retrieval	26
3.3.1 Scoring	26
3.3.2 1st naive attempt: The Jaccard coefficient	27
3.3.3 Term frequency	27
3.3.4 Document frequency	29
3.3.5 Effect of idf on ranking	30
3.4 tf-idf weighting	30
3.4.1 1st naive query implementation : simply use tf-idf for ranking documents	31
3.4.2 Weight matrix	31
3.4.3 Consider documents as vectors ...	31
3.4.4 Consider queries as vectors as well ...	32
3.4.5 Formalize vector space similarity	32
3.5 Use angle to rank document	33
3.5.1 From angle to cosine	33
3.5.2 length normalization	33
3.5.3 Compare query and documents - rank documents	34
3.5.4 Cosine example	35
3.6 General tf-idf weighting	37
3.6.1 Components of tf-idf weighting	37
3.6.2 Computing cosine scores	38
3.6.3 Conclusions	39
3.7 Practice	40
3.7.1 idf and stop words	40
3.7.2 idf logarithm base	40
3.7.3 Euclidean distance and cosine	41
3.7.4 Vector space similarity	42
3.7.5 Request and document similarities	42
3.7.6 Various questions ...	43

3.1 The Problem with Boolean search

- 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. Also good for applications: Applications can easily consume 1000s of results.
- 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 walk through 1000s of results.
 - This is particularly true of web search.

3.2 Feast of famine

The problem with boolean search can be considered as "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.

3.3 Ranked retrieval

The basis of ranked retrieval is scoring.

3.3.1 Scoring

- 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".

Query-document matching scores, the principle:

- We need a way of assigning a score to a query/document pair
- Let's start with a one-term query:
 - If the query term does not occur in the document: score should be 0
 - The more frequent the query term in the document, the higher the score (should be)
- We will look at a number of alternatives for this.

3.3.2 1st naive attempt: The Jaccard coefficient

- The Jaccard coefficient is *a commonly used measure of overlap of two sets A and B*
- $jaccard(A, B) = |A \cap B| / |A \cup B|$
- $jaccard(A, A) = 1$
- $jaccard(A, B) = 0$ if $A \cap B = \emptyset$
- Notes:
 - A and B don't have to be the same size.
 - Always assigns a number between 0 and 1.

3.3.2.1 Scoring example:

What is the query-document match score that the Jaccard coefficient computes for each of the two documents below?

- Query: "ides of march"
- Document 1: "caesar died in march" → result : 1/5
- Document 2: "the long march" → result : 1/6

3.3.2.2 Issues with Jaccard for scoring

- It doesn't consider term frequency (how many times a term occurs in a document).
- Rare terms in a collection are more informative than frequent terms. Jaccard doesn't consider this information.
- We need a more sophisticated way of normalizing for length.

3.3.3 Term frequency

3.3.3.1 Use the frequencies of terms

Recall the binary term-document incidence matrix of the previous chapter:

	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

From now on we will use the frequencies of terms in a [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

3.3.3.2 Bag of words model

- Vector representation doesn't consider the ordering of words in a document
- "John is quicker than Mary" and "Mary is quicker than John" have the same vectors
- This is called the **bag of words** model.
- In a sense, this is a step back: The positional index was able to distinguish these two documents.
- (We will be able to "recover" positional information later.)
- For now: bag of words model

3.3.3.3 Term frequency tf

- The term frequency $tf_{t,d}$ 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 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.**

3.3.3.4 Log-frequency weighting

- The log frequency weight of term t in d is :

$$w_{t,d} = \begin{cases} 1 + \log_{10} tf_{t,d}, & \text{if } tf_{t,d} > 0 \\ 0, & \text{otherwise} \end{cases}$$

One should note that this is simply the log in base 10 of the frequency + 1.

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

- Score for a document-query pair: **sum over terms** t in both q and d :
 $\text{score} : \sum_{t \in q \cap d} (1 + \log tf_{t,d})$
- The score is 0 if none of the query terms is present in the document.
- (This way we already have a first ranking using the score of each document.)

The "Score for a document-query pair" as shown above would be sufficient for a first implementation of a search system. However, it suffers from several issues that we will cover below that will make us need a more sophisticated system.

3.3.3.5 Term frequency in the query

Related to the *Term Frequency*, in addition to the count of terms in the document, we sometimes use the count of terms in the query (needed later):

$tf_{t,d}$: **tf-document** : related to the number of occurrence of the term in the **document**.
 $tf_{t,q}$: **tf-query** : related to the number of occurrence of the term in the **query**.

3.3.4 Document frequency

- **Raw term frequency as suggested above suffers from a critical problem: all terms are considered equally important when it comes to assessing relevancy on a query.** The most scarce terms are not given a higher value.
- In fact, certain terms have little or no discriminating power in determining relevance, e.g. *is, n', and, or, as, I*, etc.
- For example, a collection of documents on the car industry is likely to have the term **car** in almost every document.
- **We will use the **document frequency(df)** to capture this into computing the matching score.**

3.3.4.1 idf weight

- df_t is the **document frequency of t: the number of documents that contain t**
 - df_t is an inverse measure of the informativeness of t
 - Let N be the number of documents in the collection, then $df_t \leq N$
- **We define the idf (inverse document frequency) weight of t by :**

$$\text{idf}_t = \log_{10} (N/df_t)$$

■ We use $\log (N/df_t)$ instead of N/df_t to "dampen" the effect of idf.

Will turn out the base of the log is immaterial.

- **So we use the log transformation both for term frequency and document frequency**

3.3.4.2 Examples for idf

Suppose $N = 1'000'000$:

(There is one idf value for each term t in a collection:)

$$idf_t = \log_{10}(N/df_t)$$

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

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

3.3.5 Effect of idf on ranking

- idf affects the ranking of documents for queries with at least two terms
- For example, in the query "*arachocentric line*", idf weighting increases the relative weight of "*arachocentric*" and decreases the relative weight of "*line*"
- idf has no effect on ranking for one term queries.
- Look at **the** in the example above, idf_t for a term that apperars in every document is 0 $\Rightarrow$ the weight of this term will be 0 in the query $\Rightarrow$ its as if the term was never actually given $\Rightarrow$ equivalent as using a stop word

3.4 tf-idf weighting

The idea is now to combine the definitions of term frequency and inverse document frequency, 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*:

$$w_{t,d} = (1 + \log tf_{t,d}) \times (\log_{10}(N/df_t))$$

- 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
- Increases with the number of occurrences within a document
- Increases with the rarity of the term in the collection
- The tf-idf weighting is a key component of several indexation systems.

The tf.idf is a center concept in what we will see now. We will first cover the next concepts required for understanding the final solution we will be keeping for matching and ranking document to queries.

3.4.1 1st naive query implementation : simply use tf-idf for ranking documents

This first idea consists of using the tf.idf for ranking document for a query.

$$Score(q, d) = \sum_{t \in q \cap d} tf.idf_{t,d}$$

This is called the *overlap score measure*. A too simple yet working solution could use this score to rank results without any further step. But this is too simple to be efficient.

3.4.2 Weight matrix

How to store the weight computed for each term x document ?

Binary → Count → weight matrix

We have first tried binary matrix. Then we tried a count matrix. Now we are facing a 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 $\in \mathbb{R}$

Independently of the implementation (which exceeds to scope of this document), we will consider this matrix as a base for the rest of the lecture.. We will also consider the vectors formed by the rows and the columns.

3.4.3 Consider documents as vectors ...

Try to represent this matrix as a set set of vectors for each document:

- So we have a $|V|$ -dimensional vector space
- **Terms are axes of the space.** V terms means V dimensions

- Documents are points or vectors in this space
- Very high-dimensional: tens of millions of dimensions when you apply this to a web search engine
- These are very sparse vectors - most entries are zero.

3.4.4 Consider queries as vectors as well ...

- Key idea 1 : Do the same for queries: represent them as vectors in the space
- Key idea 2 : Rank documents according to their proximity to the query in this space
 - proximity = similarity of vectors
 - proximity $\approx$ inverse of distance
- Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model.
- Instead: rank more relevant documents higher than less relevant documents.

So having a vector for the query and vectors for the document, we are able to perform comparisons on these vectors. This is at the root of the way we will be able to rank document and retrieve more of them according to some comparison result instead of only being able to return some of them based on whether there in a set or not.

Here we might be able to return a document not having a term of the query before a document having all of them if the first document *answers* the query better than the second despite the missing term.

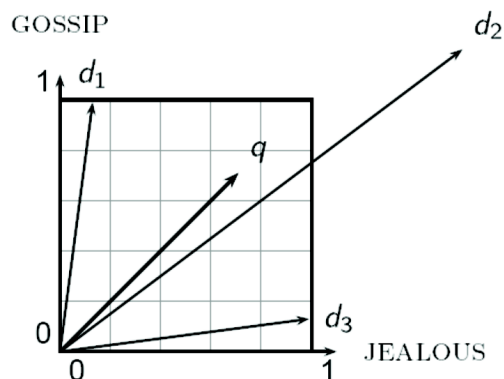
3.4.5 Formalize vector space similarity

The whole problem is to find an efficient way to formalize the vector space proximity.

3.4.5.1 Naive (bad) idea : use euclidean distance

- 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.

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.



In the above graphic we can see that 1 and d2 are good matches as the distribution of terms is close. But the euclidean distance between them is huge. We have to find something else...

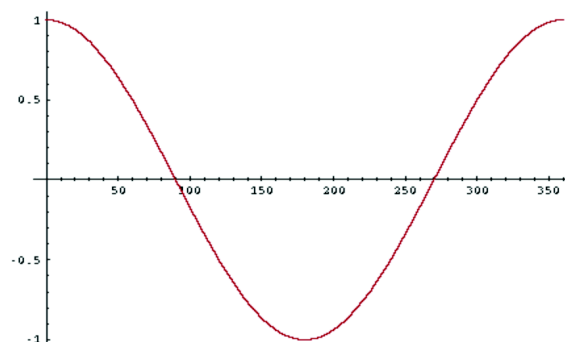
3.4.5.2 Solution: Use angle instead of distance

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

3.5 Use angle to rank document

3.5.1 From angle to cosine

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



But how – *and why* – should we be computing cosines?

3.5.2 length normalization

For comparing vectors, we're much better off comparing **unit vectors** (vecteur unitaire). A unit vector is the vector divided by its norm. As we will see in the next section, we will need unit vectors to efficiently compute the cosine.

- A vector can be (length-) normalized by dividing each of its components by its length - for this we use the L_2 norm:

$$\|\vec{x}\|_2 = \sqrt{\sum x_i^2}$$

- Dividing a vector by its L_2 norm makes it a unit (length) vector (maps it on surface of unit hypersphere)
- Effect on the two documents d and d' (d appended to itself) from earlier slide: they have identical vectors after length-normalization.
- Long and short documents now have comparable weights

...

3.5.3 Compare query and documents - rank documents

Now how to compare the query and the document and how to rank them ?

We compute the cosine similarity between the query and documents : **co-sine(query,document)**.

3.5.3.1 Principle

For each document, we compute the cosine similarity:

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

Dot product
Unit vectors

$\cos(\vec{q}, \vec{d})$ = cosine similarity of $\vec{q}$ and $\vec{d}$ = cosine of angle between $\vec{q}$ and $\vec{d}$.

where:

- 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
- $\|\vec{q}\|$ and $\|\vec{d}\|$ are the lengths of $\vec{q}$ and $\vec{d}$

3.5.3.2 Practical calculation

In practice, the calculation will be a little easier as we will manipulate length-normalized vectors. So we will first reduce the vectors to their unit vectors.

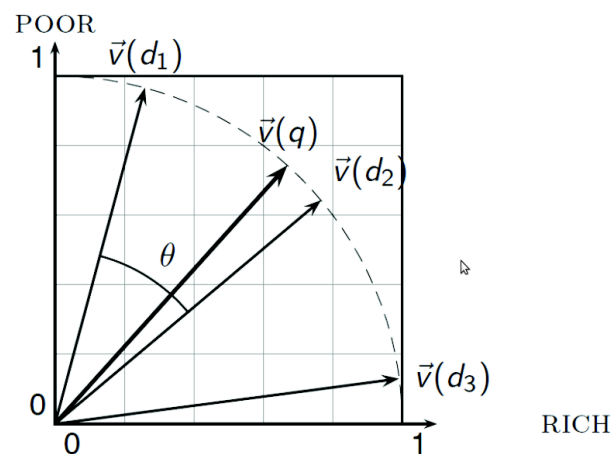
For length-normalized vectors, cosine similarity is simply the dot product (or scalar product):

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

for q, d length-normalized.

3.5.3.3 Cosine similarity illustrated

One can see the effect of the *length-normalisation* (unit vectors) and the angle between two vectors on the following graph:



3.5.4 Cosine example

Let's compute cosine similarity amongst three documents. How similar are these three novels :

SaS: Sense and Sensibility
PaP: Pride and Prejudice
WH: Wuthering Heights

What we do here is compute cosine similarity against the document themselves, no query is taken into consideration.

We want to compute :

- $\cos(\text{SaS}, \text{PaP})$
- $\cos(\text{SaS}, \text{WH})$
- $\cos(\text{PaP}, \text{WH})$

3.5.4.1 Exemple Data

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6
wuthering	0	0	38

To simplify this weighing, *we don't take any idf into consideration*. We do however *log scaling* and cosine normalisation. As we will see later, this is identified as **Inc**.

3.5.4.2 log-frequency weighting

On each cell, we apply $x = 1 + \log_{10}(x)$

term	SaS	PaP	WH
affection	3.06	2.76	2.3
jealous	2.0	1.85	2.04
gossip	1.3	0	1.78
wuthering	0	0	2.58

3.5.4.3 Compute vector norm

Reminder: we consider here document vectors (no query vectors). Hence :

- $\|\vec{Sas}\| = \sqrt{3.06^2 + 2.0^2 + 1.3^2 + 0^2} = 3.88$
- $\|\vec{PaP}\| = \sqrt{2.76^2 + 1.85^2 + 0^2 + 0^2} = 3.22$
- $\|\vec{WH}\| = \sqrt{2.3^2 + 2.04^2 + 1.78^2 + 2.58^2} = 4.39$

Now we divide each dimension of the vectors (that is each cell) by the corresponding vector norm in order to have unit vectors.

Having unit vectors help us compute cosine : With unit vectors, the cosine of the angle between the two vectors is simply the scalar product between the two vectors.

term	SaS	PaP	WH
affection	0.789	0.832	0.524
jealous	0.515	0.555	0.465
gossip	0.335	0	0.405
wuthering	0	0	0.588

3.5.4.4 Compute cosine similarities between document

We're left with a simple scalar product between the document vectors to compute similarities:

- $\cos(SaS, PaP) = 0.789 \times 0.832 + 0.515 \times 0.555 + 0.335 \times 0 + 0 \times 0 = 0.94$
- $\cos(SaS, WH) = \dots = 0.79$
- $\cos(PaP, WH) = \dots = 0.69$

One should not the highest similarity is between SaS and PaP which is quite well confirmed by the data themselves where we can see that, for instance, both doesn't contain the term "wuthering".

3.6 General tf-idf weighting

3.6.1 Components of tf-idf weighting

This graphic presents a notation for tf-idf weighting. It actually has many variants:

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

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

Note: idf is always applied on the query only. Because a term is more scarce in the documents, we want to "favorise" it in the query.

3.6.1.1 tf-idf example: Inc.Itc

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 1 : what is N, the number of documents ?

We know $idf = \log_{10}(N/df)$, hence $2.0 = \log_{10}(N/10'000)$ and hence $N = 1'000'000$.

Exercise 2 : Compute doc length = norm of document vector :

the norm is computed with the wt (weight - tf-idf) column of course

$$\|\vec{doc}\| = \sqrt{1^2 + 0^2 + 1^2 + 1.3^2} = 1.92$$

3.6.2 Computing cosine scores

Note: the weight matrix is this time attacked from the rows (vector of postings) for each term.

COSINE_SCORES(q)

```

1  float Scores[N] = 0
2  float Length[N]
3  for each query term t do # walk through terms of query
4      calculate  $w_{t,q}$  and fetch postings list for t
5      for each pair(d,  $tf_{t,d}$ ) in postings list do # walk through documents
6          Scores[d] +=  $w_{t,d} \times w_{t,q}$ 
7  Read the array Length = norms of each document
8  for each d do # walk through documents
9      Scores[d] = Scores[d] / Length[d]
10 return Top K components of Scores[] # = documents with the most weight

```

3.6.2.1 Simple example : nnn.nnn

Let's imagine the query "computer science" in the following count matrix

term	d1	d2	d3	d4	d5	d6	d7
...	...	...	...	...	...	...	...
database	1	0	0	0	5	0	7
computer	0	2	0	4	0	0	0
...	...	...	...	...	...	...	...
science	0	0	0	4	0	6	0
...	...	...	...	...	...	...	...

In this simple example we won't consider idf weighting on the query terms.

1. Compute scores

docID	$\sum tf_{query} \times tf_{doc}$
1	
2	$2 \times 1 = 2$
3	
4	$4 \times 1 + 4 \times 1 = 8$
5	
6	$6 \times 1 = 6$
7	

2. Compute length

We'll let's take random values as we cannot compute norm of document vectors without full vector definition (values for each dimension) :

docID	Length
1	...
2	5
3	...
4	6
5	...
6	7
7	...

3. Adapt Scores

docID	Score = Score / Length
1	
2	$2 / 5 = 0.4$
3	
4	$8 / 6 = 1.33$
5	
6	$6 / 7 = 0.86$
7	

3.6.3 Conclusions

3.6.3.1 Remarks

- Simple, mathematically based approach
- Consider both local (tf) and global (idf) word occurrence frequencies
- [Provides partial matching and ranked results](#)
- Tends to work quite well in practice
- Allows efficient implementation for large document collections.

3.6.3.2 Problems with vector Space Model

- Missing semantic information (e.g. word sense)
- Missing syntactic information (e.g phrase structure, word order, proximity information)
- Assumption of term independence (e.g. ignores synonymy)
- Lacks control of a Boolean model (e.g. requiring a term to appear in a document)

3.6.3.3 Implementation

In practice one cannot generate on the fly the cosine between the query and all the document in the collection (obviously) so a bit of document filtering is required. We try to identify only the documents that contain at least one query keyword.

3.7 Practice

3.7.1 idf and stop words

What is the **idf** of a term that occurs in every document ? Compare this with the use of stop words lists.

$$idf = \log_{10} N / N = \log_{10} 1 = 0$$

It is 0. Hence, for a word that occurs in every document, putting in it the stop word list has the same effect as idf weighting. The word is ignored.

3.7.2 idf logarithm base

How does the base of the logarithm in the formula $idf_t = \log(N/df_t)$ affect the score calculation? How does the base of the logarithm affect the relative scores of two documents on a given query?

For any base $b > 0$:

$$\begin{aligned} idf_t &= \log_b(N/df_t) \\ &= \log_{10}(b) \times \log_{10}(N/df_t) \\ &= c \times \log(N/df_t) \end{aligned}$$

where c is a constant

The effect on $tf.idf$ of each term is :

$$\begin{aligned}
tf - idf_{t,q,b} &= tf_{t,q} \times idf_t \\
&= tf_{t,q} \times c \times \log(N/df_t) \\
&= c * tf - idf_{t,q}
\end{aligned}$$

$$\begin{aligned}
Scores(q, d, b) &= \sum_{t \in q} tf - idf_{t,q,b} \times tf - idf_{t,d} \\
&= c \times \sum_{t \in q} tf - idf_{t,d} \times tf - idf_{t,d}
\end{aligned}$$

So changing the base changes the score of a factor $c = (\log_b 10)$ for every query term.

The relative score of documents remains unaffected by changing the base.

3.7.3 Euclidean distance and cosine

One measure of the similarity of two vectors is the Euclidean distance between them:

$$\|\vec{x} - \vec{y}\| = \sqrt{\sum_{i=1}^M (x_i - y_i)^2}$$

Given a query q and documents $d_1, d_2, \dots$, we may rank the documents d_i in order of increasing Euclidean distance from q . Show that if q and the d_i are all normalized to unit vectors, then the rank ordering produced by Euclidean distance is identical to that produced by cosine similarities.

$$\begin{aligned}
\sum (q_i - w_i)^2 &= \sum q_i^2 - 2 \sum q_i w_i + \sum w_i^2 \\
&= 2(1 - \sum q_i w_i)
\end{aligned}$$

Thus :

$$\sum (q_i - v_i)^2 < \sum (q_i - w_i)^2 \Leftrightarrow 2(1 - \sum q_i v_i) < 2(1 - \sum q_i w_i) \Leftrightarrow \sum q_i v_i > \sum q_i w_i$$

3.7.4 Vector space similarity

Compute the vector space similarity between :

1. the query: "digital cameras" and
2. the document: "digital cameras and video cameras"

by filling out the empty columns in the table below. Assume $N = 10'000'000$ logarithmic term weighting (wf columns) for query and document, idf weighting for the query only and cosine normalization for the document only.

Treat *and* as stop word. Enter term counts in the tf column. $df_{digital} = 10'000$, $df_{video} = 100'000$ and $df_{cameras} = 50'000$.

What is the final similarity score ?

Solution

Caution: the above data doesn't ask for query cosine normalisation !

word	query					document			
	tf	wf	df	idf	$q_i = wf \cdot idf$	tf	wf	$d_i = \text{normalized wf}$	$q_i \cdot d_i$
digital	1	1	10,000	3	3	1	1	0.52	1.56
video	0	0	100,000	2	0	1	1	0.52	0
cameras	1	1	50,000	2.3	2.3	2	1.3	0.68	1.56

Similarity score: $1.56 + 1.56 = 3.12$.

Normalized similarity score is also correct: $3.12 / \text{length}(\text{query}) = 3.12 / 3.78 = 0.825$

3.7.5 Request and document similarities

Compute the similarity between the request *SQL tutorial* and the document *SQL tutorial and database tutorial*.

For the term weights in the request, only the logarithmic term weighting should be used, **idf is ignored**.

Pour the term weights in the document, use normalized logarithmic frequency (cosinus normalization on logarithmic term weighting), **again no idf**.

The term *and* is a stop word.

terms	Request		Document			$q_i \times d_i$
	tf	$q_i = wf$	tf	wf	$d_i = \text{norm. wf}$	
<i>database</i>	0	0	1	1	0.52	0
<i>SQL</i>	1	1	1	1	0.52	0.52
<i>tutorial</i>	1	1	2	1.3	0.68	0.68

Using the following computations:

For the normlization, one need to compute the norm of the document vector:

$$||\vec{d}|| = \sqrt{1^2 + 1^2 + 1.3^2} = \sqrt{1 + 1 + 1.69} = \sqrt{3.69} = 1.92$$

Once the $q_i \times d_i$ computed, one can compute the similarity score:

$$Score(q, d) = \sum q_i \times d_i = 0 + 0.52 + 0.68 = 1.2$$

(One should note there is no normalization performed on the query terms as it is not asked by the instructions)

3.7.6 Various questions ...

- *Why does one use the logarithm of the term frequencies and not the raw term frequencies in order to compute the weights of the terms in the document ?*

The logarithm of the frequencies is used to avoid having more frequent terms having an overwhelming importance. For instance a term 10 x more frequent is definitely not 10 x more important. The usage of the logarithm allows to emphasize this additional importance yet not too much in order not to penalize the less frequent terms.

- *In order to compute the weight of the terms in the query:*

1. *Why does one use the IDF on the query terms ?*

The IDF enables the system to take into account the rarity of the terms in the documents. The more a term is rare, the more it will be taken into account in the query when using the IDF.

2. *Is it required to use the IDF if the request contains one single term ?*

No it's absolutely useless. The weight of a single term cannot not be more or less important to other terms when there is not other term.

- *What is precisely the advantage of using the cosinus of the angle between the vectors instead of the vector products when computing the similarity between the query vector and the document vector ?*

The problem is that we don't want to take into account the length of the vectors which has an impact of the distance between vectors. We rather take into account only the proportion of the terms in the documents and hence comparing the angles between the vectors is a better solution.

- *In order to compute the similarity between the query vector and the document vector, is it correct to use the following formula:*

$$Sim(d_j, q) = \frac{\vec{d_j} \cdot \vec{q}}{|\vec{d_j}|}$$

Yes this can be correct as there is no reason to normalize the weights of the terms in the request when there is one single request.

Evaluation

Contents

4.1 Introduction	45
4.1.1 Evaluation corpus	46
4.2 Effectiveness Measures	46
4.2.1 Recall	47
4.2.2 Precision	47
4.2.3 Trade-off between Recall and Precision	47
4.2.4 Classification errors	48
4.2.5 F Measure	48
4.3 Ranking Effectiveness	49
4.3.1 Summarizing a ranking	49
4.3.2 AP - Average precision	50
4.3.3 MAP - Mean Average Precision	50
4.3.4 Recall-Precision graph	51
4.3.5 Interpolation	52
4.3.6 Average Precision at Standard Recall Levels	53
4.4 Focusing on top documents	54
4.5 Practice	54
4.5.1 Precision and Recall	54
4.5.2 Search systems comparison	55
4.5.3 Search system analysis	56
4.5.4 Search systems comparison (another example)	58
4.5.5 F-measure	59

4.1 Introduction

- Evaluation is key to building **effective** (effectiveness = performance) and **efficient** (efficiency = efficacité) search engines.
 - Measurement usually carried out in controlled laboratory
 - online testing can however also be done
- Effectiveness, efficiency and cost are related. There is a triangle relation between them.

- e.g. if we want a particular level of effectiveness and efficiency, this will increase the cost of the system.
- efficiency and cost targets impact effectiveness

4.1.1 Evaluation corpus

Evaluation corpus are test collections consisting of documents, queries and relevance judgments. These collections are assembled by researchers for evaluating search engines.

Let's just mention that there are classic famous such collections :

- CACM : Titles and abstracts from the Communications of the ACM
 - AP : Associated Press newswire documents
 - GOV2 : Web pages crawled from the websites in the .gov domains.
-
- CACM: Titles and abstracts from the Communications of the ACM from 1958-1979. Queries and relevance judgments generated by computer scientists.
 - AP: Associated Press newswire documents from 1988-1990 (from TREC disks 1-3). Queries are the title fields from TREC topics 51-150. Topics and relevance judgments generated by government information analysts.
 - GOV2: Web pages crawled from websites in the .gov domain during early 2004. Queries are the title fields from TREC topics 701-850. Topics and relevance judgments generated by government analysts.



One technique for building these document collections is *Pooling* : top k results from the rankings obtained by different search engines are produced in some random order to relevance judges.

Another solution consists in using query logs. Query logs are used both for tuning and evaluation of search engines. The principle of query logs consists in logging the query and the user clicks.

4.2 Effectiveness Measures

For a *specific query* on a specific collection, let :

- A** Be the set of relevant documents
B Be the set of retrieved documents

	Relevant	Non-Relevant
Retrieved	$A \cap B$	$\bar{A} \cap B$
Not Retrieved	$A \cap \bar{B}$	$\bar{A} \cap \bar{B}$

4.2.1 Recall

Recall is the proportion of relevant documents that are retrieved. A good recall $\Rightarrow$ we found most if not all relevant documents.

Recall answers: What fraction of the relevant documents in the collection were returned by the system ?

$$Recall = \frac{|A \cap b|}{|A|}$$

4.2.2 Precision

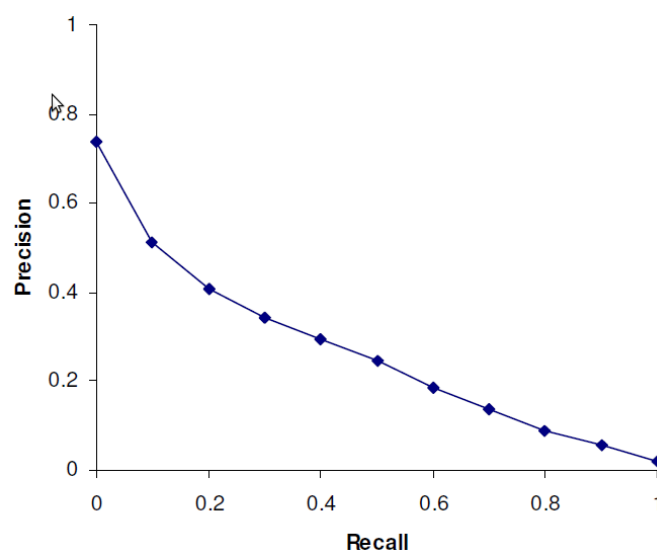
Precision is the proportion of retrieved documents that are relevant. We want as few not-relevant documents as possible in the results (only the relevant documents).

Precision answers: What fraction of the returned results are relevant to the information need?

$$Precision = \frac{|A \cap B|}{|B|}$$

4.2.3 Trade-off between Recall and Precision

There is a trade-off between recall and precision. The relation between the two measures usually is like this (this is a good system) :



Precision and Recall are inversed proportionnal $\Rightarrow$ one cannot have them both high.

4.2.4 Classification errors

We want to be able to compute classification errors. We consider therefor two measures:

4.2.4.1 False positive

A false positive is a non-relevant document retrieved. We use the *fallout* for this:

$$Fallout = \frac{|\bar{A} \cap B|}{|\bar{A}|}$$

The goal is to have a fallout as little as possible.

4.2.4.2 False negative

A false negative is about a relevant document that is not retrieved. This is simple $1 - Recall$.

4.2.5 F Measure

The *F measure* is an effectiveness measure based on recall and precision that is used for evaluating classification performance.

4.2.5.1 Harmonic mean

Harmonic mean of recall and precision :

$$F = \frac{2RP}{R + P}$$

We use the harmonic mean instead of the arithmetic means because the harmonic mean emphasizes the importance of small values, whereas the arithmetic mean is affected more by outliers that are unusually large.

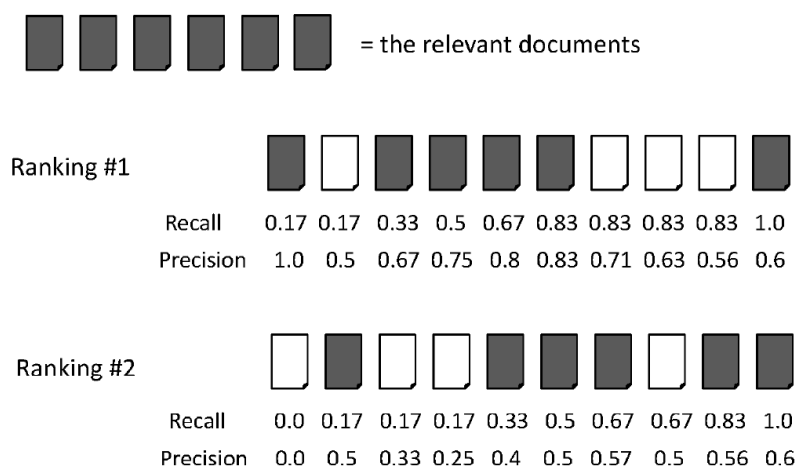
4.2.5.2 General Form

The general form of the F Measure is the following :

$$F_{\beta} = \frac{(\beta^2 + 1)RP}{R + \beta^2 P}$$

4.3 Ranking Effectiveness

We have here two different ranking systems which we want to evaluate. The graphic shows the 10 first results. There were 6 relevant documents in total in the collection.



At each position in the ranked results, we compute precision and recall at that precise step/-moment.

→ These values are called *recall or precision at rank p*.

The problem:

At rank position 10, the two rankings system have the same effectiveness as measured by precision and recall. Recall is 1.0 because all the relevant documents have been retrieved and precision is 0.6 because both rankings contain 6 relevant documents in the retrieved set of 10 documents.

At higher positions, however, the first ranking system is better, one can look for instance at position 4.

Hence:

We need a better way to compare the ranking systems than simply *precision at rank p*.

A large number of techniques have been developed to summarize the effectiveness of ranking systems. The first one is simply to calculate recall/precision values at small number of predefined rank positions.

4.3.1 Summarizing a ranking

There are three usual ways for summarizing a ranking

1. Calculating recall and precision at fixed rank positions

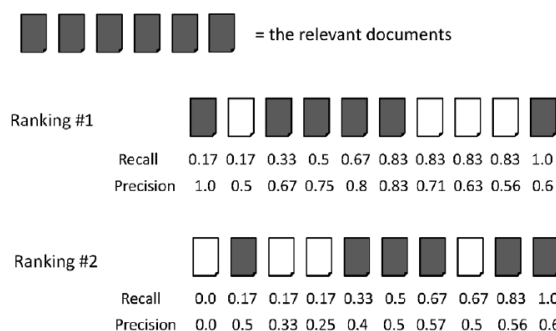
→ simply look at fixed positions, usually 10 or 20. Example : above at position 4 or 10.

2. Averaging the precision values from the rank positions where a relevant document was retrieved. This is called AP - Average Precision or MAP - Mean Average Precision for multiples queries
→ See section 4.3.2
3. Calculating precision at standard recall levels, from 0.0 to 1.0 (usually requires interpolation)
→ See section 4.3.6

4.3.2 AP - Average precision

A quite popular method is to summarize the ranking by [averaging the precision values from the rank positions where a relevant document was retrieved \(i.e. when recall increases\)](#).

This way, the more the system returns relevant document at lower ranks, the better the average precision is:



$$\text{Ranking \#1: } (1.0 + 0.67 + 0.75 + 0.8 + 0.83 + 0.6)/6 = 0.78$$

$$\text{Ranking \#2: } (0.5 + 0.4 + 0.5 + 0.57 + 0.56 + 0.6)/6 = 0.52$$

We take the sum of the precision at each rank where a relevant document is retrieved and we divide by the number of relevant documents.

For instance, in the example above, we have divided the sum of the precision of the ranks where a relevant document was retrieved and we divided it by 6 = the number of relevant documents.

4.3.3 MAP - Mean Average Precision

The aim of an averaging technique is to summarize the effectiveness of a specific ranking algorithm accross several queries, not only one as we did above.

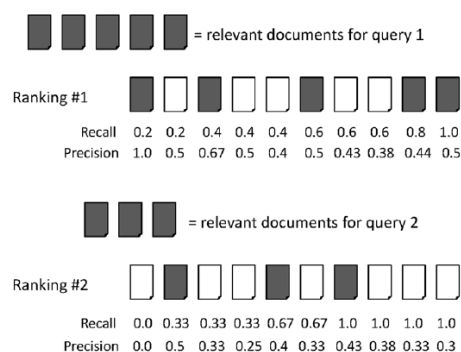
In order to achieve this, **one idea is simply to use the mean of average precision accross several queries.**

This is the MAP for *Mean Average Precision* :

- summarize rankings from multiple queries by averaging AP - Average Precision

- most commonly used measure in research papers
- assumes user is interested in finding many relevant documents for each query
- requires many relevance judgments in text collection

Note: recall precision graphs are also useful summaries.



$$\text{average precision query 1} = (1.0 + 0.67 + 0.5 + 0.44 + 0.5)/5 = 0.62$$

$$\text{average precision query 2} = (0.5 + 0.4 + 0.43)/3 = 0.44$$

$$\text{mean average precision} = (0.62 + 0.44)/2 = 0.53$$

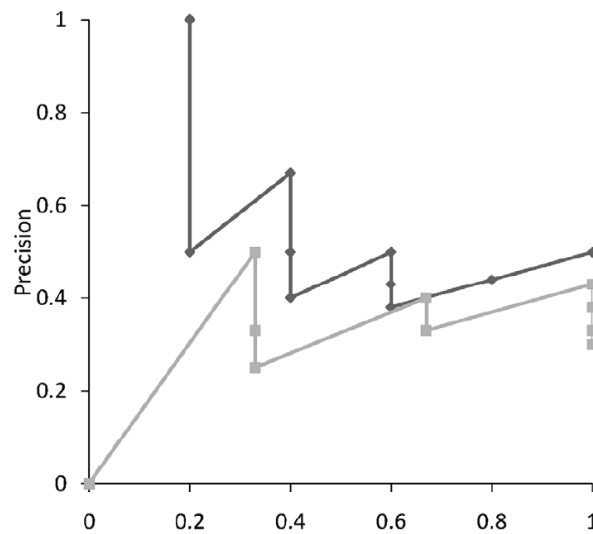
The goal is still to see how much the system is good at returning the relevant document in the head of the result list, but accross several queries. The MAP provides a very succinct summary of the effectiveness of the ranking algorithm.

Note: sometimes AP (single query) is called MAP as well.

4.3.4 Recall-Precision graph

However, sometimes too much information is lost in this process. [Recall precision graphs, and the recall-precision tables they are based on, give more detail on the effectiveness of the ranking algorithm.](#)

This graph below is a Recall-Precision graph for the above example:



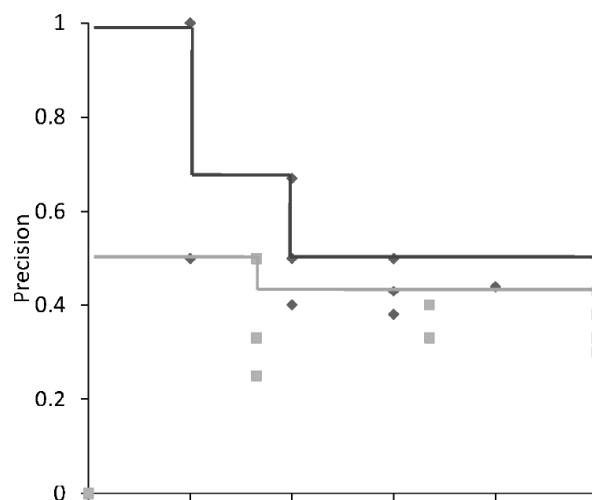
4.3.5 Interpolation

- To average graphs, calculate precision at standard recall levels:

$$P(R) = \max\{P' : R' \geq R \wedge (R', P') \in S\}$$

where S is the set of observed (R,P) points

- Defines precision at any recall level as the maximum precision observed in any recall-precision point at a higher recall level
- produces a step function
- defines precision at recall 0.0
- At any recall level (x-axis) we take the maximum precision value available at this level or any higher level.



This interpolation method is consistent with the observation that it produces a function that

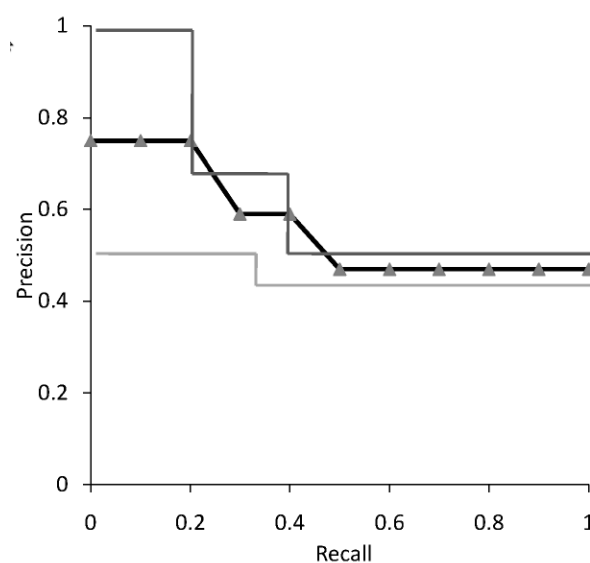
is monotonically decreasing. This means that precision value always goes down (or stays the same) with increasing recall.

It also defines a precision for recall 0.0 which would not be obvious otherwise.

4.3.6 Average Precision at Standard Recall Levels

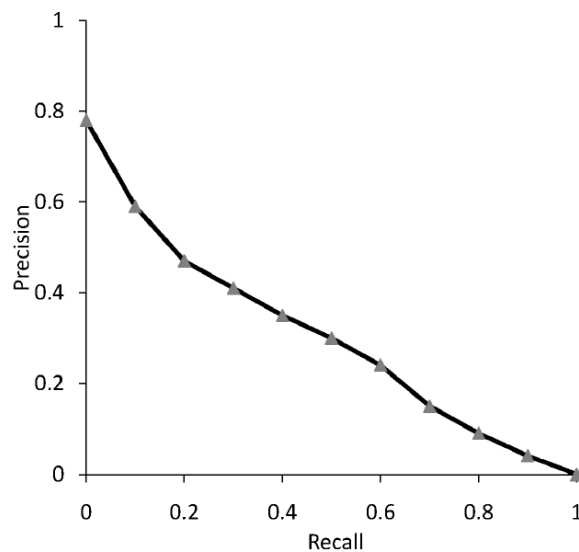
Now we know how to interpolate the results, we can average the interpolated values. This implies plotting the Recall-precision graph by simply joining the average precision points at the **standard recall levels**.

The standard precision levels are 0.1, 0.2, 0.3, 0.4, ...



Although this is somewhat inconsistent with the interpolation method, the intermediate recall levels are never used in evaluation.

When graphs are averaged over many queries, they tend to become smoother:



How to compare 2 systems ? : the curve closest to the upper right corner of the graph is the best performance.

4.4 Focusing on top documents

- Users tend to look at only the top part of the ranked result list to find relevant documents
- Some search tasks have only one relevant documents, e.g. navigational search, question answering
- Recall not appropriate → instead need to measure how well the search engine does at retrieving relevant documents at very high ranks

Precision at rank R:

- R typically 5, 10, 20
- easy to compute, average, understand
- not sensitive to rank positions less than R

R-Precision is the precision at the R-th position in the ranking of results for a query that has *R* relevant documents.

4.5 Practice

4.5.1 Precision and Recall

An IR system returns 8 relevant documents and 10 non-relevant documents. There are a total of 20 relevant documents in the collection. What is the precision of the system on this search, and what is its recall ?

$$A = |\text{relevant documents}| = 20$$

$$B = |\text{retrieved documents}| = 18$$

$$\text{Recall} = \frac{|A \cap B|}{|A|} = \frac{8}{20} = 0.4$$

$$\text{Precision} = \frac{|A \cap B|}{|B|} = \frac{8}{18} = 0.45$$

4.5.2 Search systems comparison

Consider an information need for which there are 4 relevant documents in the collection. Contrast two systems run on this collection. Their top 10 results are judged for relevance as follows (the leftmost item is the top ranked search result):

System 1 RNRNN NNNRR
System 2 NRNNR RRNNN

4.5.2.1 MAP

What is the MAP of each system ? Which has a higher MAP ?

System 1	R	N	R	N	N	N	N	N	R	R
→ Recall	0.25	0.25	0.5	0.5	0.5	0.5	0.5	0.5	0.75	1
→ Precision	1	0.5	0.67	0.5	0.4	0.33	0.29	0.25	0.33	0.4
System 2	N	R	N	N	R	R	R	N	N	N
→ Recall	0	0.25	0.25	0.25	0.5	0.75	1	1	1	1
→ Precision	0	0.5	0.33	0.25	0.4	0.5	0.57	0.5	0.44	0.4

$$AP_1 = \frac{1 + 0.67 + 0.33 + 0.4}{4} = 0.6$$

$$AP_2 = \frac{0.5 + 0.4 + 0.5 + 0.57}{4} = 0.49$$

So system 1 has a higher MAP.

4.5.2.2 MAP analysis

Does this intuitively make sense ? What does it say about what is important in getting a good MAP score ?

This does actually make sense as it is more important for the MAP (and for a good search engine) to have the relevant results reported as early as possible. Having the first result a relevant one is very important.

4.5.2.3 R-Precision

What is the R-precision of each system ? (Does it rank the systems such as the MAP?)

There are 4 relevant document $\Rightarrow R = 4$

$$R_1 = \frac{2}{4} = 0.5$$

$$R_2 = \frac{1}{4} = 0.25$$

So the tendency is pretty much the same than with the MAP.

4.5.3 Search system analysis

The following list of R's and N's represents relevant (R) and non-relevant (N) documents returned in a ranked list of 20 documents retrieved in response to a query on a collection of 10'000 documents.

The top of the ranked list (the documents the system thinks is most likely to be relevant) is on the left or the list.

This list shows 6 relevant document. Let's assume there are 8 relevant documents in total in the collection.

RRNNN NNNRN RNNNR NNNNR

4.5.3.1 Precision and Recall

<i>(partie 1)</i>	R	R	N	N	N	N	N	N	R	N
→ Recall	0.13	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.38	0.38
→ Precision	1	1	0.67	0.5	0.4	0.33	0.29	0.25	0.33	0.3

<i>(partie 2)</i>	R	N	N	N	R	N	N	N	N	R
→ Recall	0.5	0.5	0.5	0.5	0.63	0.63	0.63	0.63	0.63	0.75
→ Precision	0.36	0.33	0.31	0.29	0.33	0.31	0.29	0.28	0.26	0.3

What is the precision of the system on the top 20 ?

$$Recall = \frac{|A \cap B|}{|A|} = \frac{6}{8} = 0.75$$

$$Precision = \frac{|A \cap B|}{|B|} = \frac{6}{20} = 0.3$$

4.5.3.2 What is the F1 on the top 20 ?

The F_1 is the F Measure for $\beta = 1$

$$F_\beta = \frac{(\beta^2 + 1)RP}{R + \beta^2 P}$$

$$F_1 = \frac{2RP}{R + P} = \frac{2 \times 0.75 \times 0.3}{0.75 + 0.3} = \frac{0.45}{1.05} = 0.43$$

4.5.3.3 Uninterpolated precision

What is the uninterpolated precision of the system at 25% recall ?

25% recall means 2 relevant documents out on 8. Here at 0.25 recall we have several values for the precision (from 1.0 at rank 2 to 0.25 at rank 8), we take the highest one which is necessarily the precision right at the moment the 25% recall has been reached

The answer is : 1.0.

4.5.3.4 Interpolated precision

What is the interpolated precision of the system at 33% recall ?

33% recall means 2.64 relevant documents out on 8.

Interpolated $\Rightarrow$ we take the maximum one from the precision values of the recall points above 0.33.

Looking on the array above, the highest precision we find for a recall above 0.33 is at rank 11 where the precision is 0.36 for a recall of 0.5.

The answer is : 0.36.

4.5.3.5 MAP

Assume that these 20 documents are the complete result set of the system. What is the MAP for the query ?

$$AP = \frac{1 + 1 + 0.33 + 0.36 + 0.33 + 0.3}{8} = 0.41$$

Note the division by 8 = total relevant documents in the collection and not 6 = relevant documents retrieved.

4.5.3.6 Largest MAP

What is the largest possible MAP that this system could have ?

We assume the 20 first results are fixed. The largest possible MAP occurs when the 2 remaining relevant documents appear at the earliest ranks, that is 21 and 22. In this case:

$$AP = \frac{1 + 1 + 0.33 + 0.36 + 0.33 + 0.3 + \frac{7}{21} + \frac{8}{22}}{8} = 0.50$$

4.5.3.7 Smallest MAP

What is the smallest possible MAP that this system could have ?

We assume the 20 first results are fixed. The smallest possible MAP occurs when the 2 remaining relevant documents appear at the latest ranks, that is 19'999 and 20'000. In this case:

$$AP = \frac{1 + 1 + 0.33 + 0.36 + 0.33 + 0.3 + \frac{7}{19'999} + \frac{8}{20'000}}{8} = 0.41$$

4.5.4 Search systems comparison (another example)

One wants to compare the performance of two search systems A and B on a specific request R. The results of the search engines sorted by relevance are given below (R is a relevant result while N is a non-relevant result):

System A	RRRRN	NNRNN
System B	RRNNR	NNNRR

We know there is a total of 10 relevant documents in the collection. One should not that both systems have returned only 5 relevant documents.

However one of them is deemed as more performant than the other.

- According to you, which is the most performant one ?
- Then, justify your answer using an appropriate metric.

The first system is more efficient as it favorise relevant documents at the beginning of the result list.

System A	R	R	R	R	N	N	N	R	N	N
→ Recall	0.1	0.2	0.3	0.4	0.4	0.4	0.4	0.5	0.5	5
→ Precision	1	1	1	1	0.8	0.67	0.58	0.63	0.55	0.5
System B	R	R	N	N	R	N	N	N	R	R
→ Recall	0.1	0.2	0.2	0.2	0.3	0.3	0.3	0.3	0.4	0.4
→ Precision	1	1	0.67	0.5	0.6	0.5	0.43	0.38	0.44	0.5

One can compute for instance R and P and rank 5:

$$R_{1-r5} = 0.4$$

$$P_{1-r5} = 0.8$$

$$R_{2-r5} = 0.3$$

$$P_{2-r5} = 0.6$$

Or perhaps the (Mean) Average Precision:

$$AP_1 = \frac{1 + 1 + 1 + 1 + 0.63}{10} = 0.46$$

$$AP_2 = \frac{1 + 1 + 0.6 + 0.44 + 0.5}{10} = 0.35$$

Both these results confirm our instinctive approach.

4.5.5 F-measure

The F-measure is defined as the hamonic mean between the Recall and Precision. What is the advantage of using the harmonic mean instead of the arithmetic mean:

We use the harmonic mean instead of the arithmetic means because **the harmonic mean emphasizes the importance of small values, whereas the arithmetic mean is affected more by outliers that are unusually large.**

Queries and Interfaces

Contents

5.1 Introduction	61
5.1.1 Information Needs	61
5.1.2 Queries and Information Needs	62
5.1.3 Interaction	62
5.1.4 ASK Hypothesis	62
5.2 Query-based Stemming	62
5.2.1 Stem Classes	62
5.3 Spell checking	63
5.3.1 Basic Approach	63
5.3.2 Noisy channel model	64
5.4 Relevance feedback	64
5.4.1 Query reformulation	64
5.4.2 Optimal query	65
5.4.3 Standard Rocchio Method	65
5.5 practice	66
5.5.1 Relevance feedback	66

5.1 Introduction

5.1.1 Information Needs

- An information need is the underlying cause of the query that a person submits to a search engine.
→ sometimes called information problem to emphasize that information need is generally related to a task
- Categorized using variety of dimensions
 - e.g., number of relevant documents being sought
 - type of information that is needed
 - type of task that led to the requirement for information

5.1.2 Queries and Information Needs

- A query can represent very different information needs
 - May require different search techniques and ranking algorithms to produce the best rankings
- A query can be a poor representation of the information need
 - User may find it difficult to express the information need
 - User is encouraged to enter short queries both by the search engine interface, and by the fact that long queries don't work

5.1.3 Interaction

- Interaction with the system occurs
 - during query formulation and reformulation
 - while browsing the result
- Key aspect of effective retrieval
 - users can't change ranking algorithm but can change results through interaction
 - helps refine description of information need

5.1.4 ASK Hypothesis

- Belkin et al (1982) proposed a model called Anomalous State of Knowledge
- ASK hypothesis:
 - difficult for people to define exactly what their information need is, because that information is a gap in their knowledge
 - Search engine should look for information that fills those gaps

5.2 Query-based Stemming

- Make decision about stemming at query time rather than during indexing.
→ improved flexibility, effectiveness
- Query is expanded using word variants
 - documents are not stemmed
 - e.g., "rock climbing" expanded with "climb", not stemmed to "climb"

5.2.1 Stem Classes

We need a way to generate the stem. We define for this *stem classes* built using, for instance, the *porter* algorithm. A *stem class* is the group of words that will be transformed into the same stem by the stemming algorithm.

/bank banked banking bankings banks
 /ocean oceaneering oceanic oceanics oceanization oceans
 /polic polical polically police policeable policed
 -policement policer policers policies policial
 -policically policier policiers policies policing
 -policization policize policly policy policying policys

As one can see there is a problem in the example above as *policy* and *police* should not necessarily be grouped together.

- Stem classes are often too big and inaccurate
- Modify using analysis of word co-occurrence
- **Assumption:** Word variants that could substitute for each other should co-occur often in documents.

1. For all pairs of words in the stem classes, count how often they co-occur in text windows of W words. W is typically in the range 50-100.
2. Compute a co-occurrence or association metric for each pair. This measures how strong the association is between the words.
3. Construct a graph where the vertices represent words and the edges are between words whose co-occurrence metric is above a threshold T .
4. Find the connected components of this graph. These are the new stem classes.

5.3 Spell checking

- Spell checking Important part of query processing
- 10-15% of all web queries have spelling errors
- Errors include typical word processing errors but also many other types, typo, syntax, etc.

5.3.1 Basic Approach

The idea consist in suggesting corrections for words not found in spelling dictionary.

- Suggestions found by comparing word to words in dictionary using similarity measure
- Most common similarity measure is **edit distance** : the number of operations required to transform one word into the other

5.3.1.1 Edit Distance

Most commonly, the *Damerau-Levenshtein* distance is used: counts the minimum number of insertions, deletions, substitutions, or transpositions of single characters required.

- Ex. for distance = 1:
 extensions → extensions / (insertion error)
 pointter → pointer / (deletion error)
 painter → pointer / (substitution error)
 poniter → pointer / (transposition error)
- Ex. for distance = 2:
 doceration → decoration

There are a number of techniques used to speed up calculation of edit distances:

- restrict to words starting with same character
- restrict to words of same or similar length
- restrict to words that sound the same

The last option uses a *phonetic code* such as the Soundex (see lecture).

5.3.2 Noisy channel model

The noisy channel model is based on Shannon's theory of communication. I won't cover it here in this resume.

5.4 Relevance feedback

- consists in getting user feedback on relevance of documents in the initial set of results:
 1. User issues a (short, simple) query
 2. The user marks from results as relevant or non-relevant
 3. The system computes a better representation of the information need based on this feedback
 4. Relevance feedback can go through one or more iterations
- Idea : it may be difficult to formulate a good query when you don't know the collection well, so iterate → make the system learn.

5.4.1 Query reformulation

- Revise query to account for feedback:
 1. **Query expansion** : Add new terms to query from relevant documents.
 2. **Term reweighting** : Increase weight of terms in relevant documents and decrease weight of terms in irrelevant documents.

5.4.1.1 Query reformulation in Vector model

- Change query vector using vector algebra
- Add the vectors for the relevant document to the query vector

- Subtract the vectors for the irrelevant docs from the query vector
- This adds both positively and negatively weighted terms to the query as well as reweighting the initial query terms.

5.4.2 Optimal query

Assume that the relevant set of documents C_r is known. Then the best query that ranks all and only the relevant documents at the top is:

$$q_{opt} = \underbrace{\frac{1}{|C_r|} \sum_{\forall d_j \in C_r} \vec{d}_j}_A - \underbrace{\frac{1}{N - |C_r|} \sum_{\forall d_j \notin C_r} \vec{d}_j}_B$$

where:

C_r	is the set of relevant documents in the collection.
N	is the total number of documents
$N - C_r $	is the number of irrelevant documents
A	is the centroid of the cluster of the relevant documents
B	is the centroid of the cluster of the irrelevant documents

5.4.3 Standard Rocchio Method

Since all the set of all relevant documents is unknown in practice, we need to find a slightly different formula a search engine could use.

The idea is to use instead the set of **known** relevant documents (D_r) and the set of **known** irrelevant documents (D_n) and include the initial query q in the formula:

$$\vec{q}_m = \alpha \vec{q} + \frac{\beta}{|D_r|} \sum_{\forall d_j \in D_r} \vec{d}_j - \frac{\gamma}{|D_n|} \sum_{\forall d_j \in D_n} \vec{d}_j$$

where:

α	is the tunable weight for initial query
β	is the tunable weight for relevant documents
γ	is the tunable weight for irrelevant documents

Usual, one chooses a β greater than the γ value in order to give a higher weight to the relevant documents.

The resulting negative weights are set to 0.

5.5 practice

5.5.1 Relevance feedback

Let's assume an initial request of a user such as:

"cheap CDs cheap DVDs extremely cheap CDs"

Then the user examines 2 documents D_1 and D_2 :

- D_1 : *CDs cheap software cheap CDs*
- D_2 : *cheap thrill DVDs*

which the user judges as D_1 is relevant and D_2 is irrelevant.

- Use direct term frequencies, no scaling, no idf, no normalization.
- Use Rocchio's relevance
- Assume $\alpha = 1$, $\beta = 0.75$, $\gamma = 0.25$
- What would be the revised query vector after relevance feedback ?

Computing vectors:

	Query		D_1		D_2		$\vec{q}_m$
	init.	$\alpha \vec{q}$	init.	$\beta \vec{D}_1$	init.	$\gamma \vec{D}_2$	
CD	2	2	2	1.5	0	0	$2 + 1.5 - 0 = \mathbf{3.5}$
cheap	3	3	2	1.5	1	0.25	$3 + 1.5 - 0.25 = \mathbf{4.25}$
DVD	1	1	0	0	1	0.25	$1 + 0 - 0.25 = \mathbf{0.75}$
extremely	1	1	0	0	0	0	$1 + 0 - 0 = \mathbf{1}$
software	0	0	1	0.75	0	0	$0 + 0.75 - 0 = \mathbf{0.75}$
thrills	0	0	0	0	1	0.25	$0 - 0 - 0.25 = 0 \rightarrow \mathbf{0}$

