UNIVERSITÄT BASEL

Computer Science / 15731-01 / 2018

Multimedia Retrieval

Chapter 2: Text Retrieval

2.1 Overview and Motivation
2.2 Feature Extraction
2.3 Text Retrieval Models
2.4 Indexing Structures
2.5 Lucene - Open Source Text Search
2.6 Literature and Links

Dr. Roger Weber, roger.weber@ubs.com

2.1 Overview and Motivation

- Managing and retrieving information remains a challenging problem despite the impressive advances in computer science. The first generation of computers used punch cards to store and retrieve information, and memory and compute was precious. Many early algorithms hence have used Boolean models and brute-force approaches that quickly decide whether something is relevant or not. Today, memory and compute are extremely cheap, and we added more elaborated retrieval techniques to speed up the searches. Only recently, map-reduce and deep learning have gone back to the brute-force methods of the early days.
- Typical types of information retrieval:
  - **Database:** information is maintained in a structured way. Queries refer to the structure of the data and define constraints on the values (SQL as query language). Being structured, however, does not allow for quick retrieval across all data items with something like this:

    ```
    SELECT * FROM * WHERE * like '%house%'
    ```

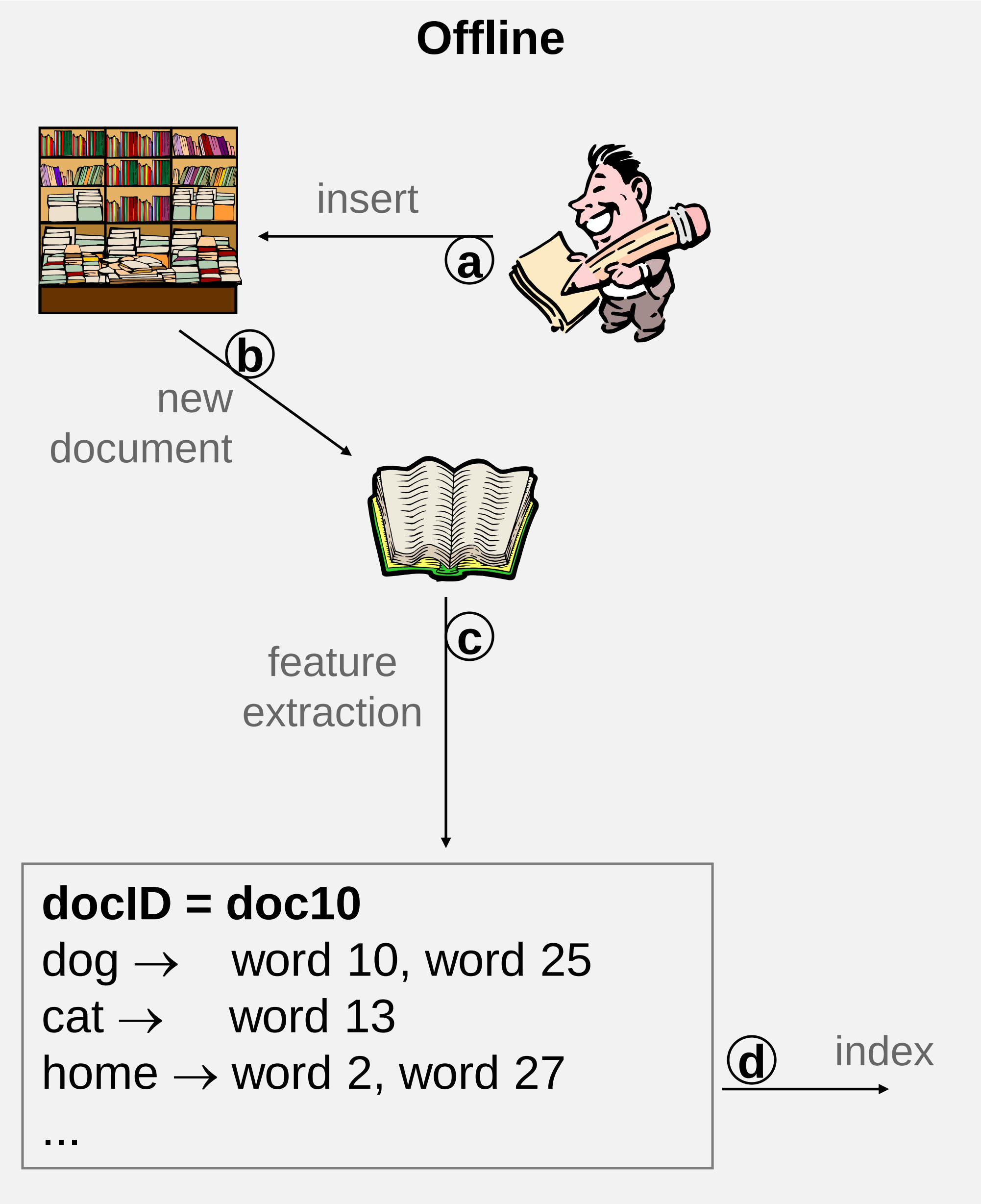
  - **Boolean Retrieval Systems:** Boolean models simplified matters: while scanning the data, we can decide whether an entry is relevant or not. There is no need to keep track and sort results later on. This was a huge advantage for early information systems (those with the punch cards and later with tapes) as they only had to filter out which data items were relevant based on a Boolean outcome. Even though rather simple, it is still a dominant retrieval model.
  - **Retrieval System with Ranking:** Basic Boolean retrieval suffers from the lack of a ranked list. A user is typically interested in a few, good answers but has not the time to go through all of the potential thousands of relevant documents. If you search a book in an online store, you expect the best matches to be at the top. Newer models, hence, try to determine how relevant a document is for the user (in his given context) given the query.

- **Vague Queries against Database:** this search type allows the user to specify soft constraints, i.e., vague query parts. For instance, if you want to buy a new computer, you may specify an “Intel Core i7” CPU, 32GB of memory, 1TB of SSD, and at least GTX-980 graphics card. And of course, you don’t want to pay more than $1000. As you walk through the options, you may realize that you can’t satisfy all constraints and you compromise on some of them (e.g., replace SSD with HDD but now with 4TB). Vague queries are best executed with “fuzzy” retrieval models with a cost function that needs to be optimized (to satisfy the user’s demand as far as possible)
- **Natural Language Processing (NLP):** Consider a database with industrial parts for machines. A complex query may look as follows:
  - **“**Find bolts made of steel with a radius of 2.5 mm, a length of 10 cm implementing DIN 4711. The bolts should have a polished surface and can be used within an electronic engine.”

  The challenge of the above query is that we are not actually looking for the keywords “radius”, “DIN”, or “polished”. Rather, the keywords refer to constraints and to a context expressed by the user. Recent improvements in Natural Language Processing (NLP) enabled systems to “decipher” such queries. Modern recommendation systems can chat with the user to obtain the context and then perform a search to answer the information need. We will, however, not look at such systems in this course, but lay a few foundations here and there.
- **Web Retrieval:** early (text) retrieval systems focused on searches over managed and controlled document collections. With the Web, search engines were faced with spamming, bad quality, aggressive advertisements, fraud, malware, and click baits. Many retrieval models failed completely in this uncontrolled environment. Web retrieval addresses many of these concerns and tries to find, among trillions of possible answers, the best few pages for your query. The sheer volume of information is a challenge in its own.

- **Multimedia Content:** with cheap storage and the digital transformation of enterprises and consumers, enormous amounts of multimedia data gets created every day (images, audio files, videos). The methods of text retrieval only work on the meta data but not on the signal information of the content. We still have a large **semantic gap** when searching for multimedia content, but recent improvements in deep learning techniques rapidly close that gap. These techniques automatically label multimedia content to enable simpler text (or speech) search over multimedia content and thereby bridging the semantic gap between the signal information and the user’s intent.
- **Heterogeneous, Distributed, Autonomous Information Sources:** meta search is a generic problem: the user does not want to repeat a query against all information sources, but rather search once against all systems. In more complex setups, each system may hold the answer to a part of the query and only the combination of all parts yields the best results. We will consider more complex searches in later chapters.

2.1.1 Text Retrieval – Overview



Offline

insert

a

b

new document

c

feature extraction

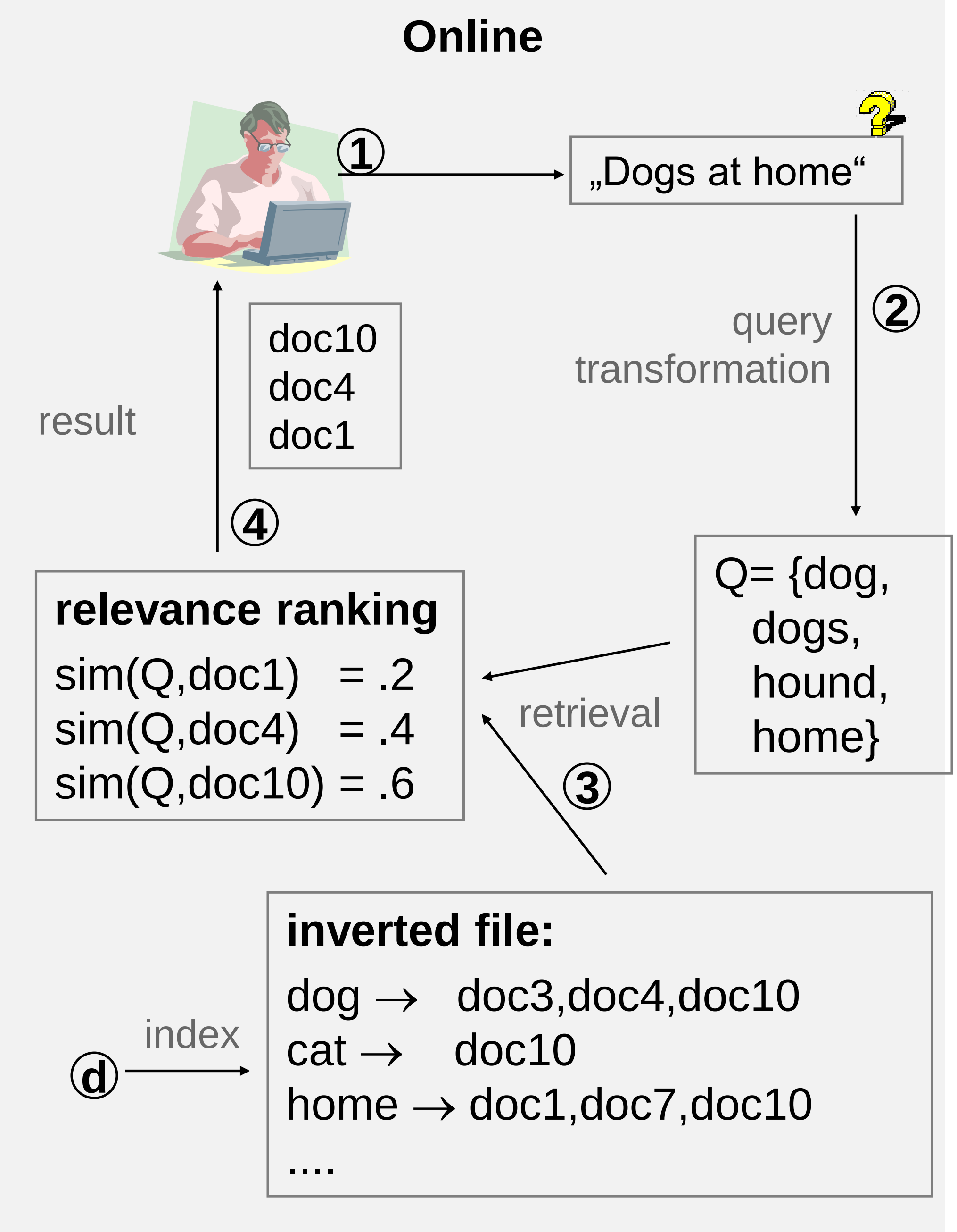
docID = doc10
dog → word 10, word 25
cat → word 13
home → word 2, word 27
...

d

index

- Text retrieval encompasses two modes:
  - an offline mode, that allows us to add documents and to analyze them, and
  - an online mode, that retrieves relevant documents for queries given by users
- Obviously, we do not want to apply text search on the native documents. Rather we extract so-called features which represent characteristic pieces of information about the content of the document. The features also should support fast retrieval afterwards.
- In more detail, the following steps occur during the offline mode:
  a) We add a new document (or we find a new document by scanning/crawling)
  b) Each addition triggers an event to extract features and update search indexes
  c) We extract features that based describe the content and analyze & reason on context and higher-level features
  d) We pass the features to an index that accelerates searches given a query

- In the online mode, users can search for documents. The query is analyzed similarly to the documents in the offline mode, but often we apply additional processing to correct spelling mistakes or to broaden the search with synonyms. The retrieval, finally, is a comparison at the feature level. We assume that two documents that have similar features also are similar in content. Hence, if the features of the query are close to the ones of the document, the document is considered a good match.
- In more detail, the following steps occur during the offline mode:
  1) User enters a query (or speech/ handwriting recognition)
  2) We extract features like for the documents, and transform the query as necessary (e.g., spelling mistakes)
  3) We use the query features to search the index for document with similar features
  4) We rank the documents (retrieval status value, RSV) and return best documents



Online

① „Dogs at home“

② query transformation

Q= {dog, dogs, hound, home}

③ retrieval

relevance ranking

sim(Q,doc1) = .2
sim(Q,doc4) = .4
sim(Q,doc10) = .6

④ result

doc10
doc4
doc1

ⓓ index

inverted file:

dog → doc3,doc4,doc10
cat → doc10
home → doc1,doc7,doc10
....

2.1.2 The Retrieval Problem

Given

- N text documents $\mathbb{D} = (D_1, \dots, D_N)$ and the Query Q of the user

Problem

- find ranked list of documents which match the query well; ranking with respect to relevance of document to the query

- We will consider the following parts of the problem in this chapter:
  - Feature extraction (words, phrases, n-grams, stemming, stop words, thesaurus)
  - Retrieval model (Boolean retrieval, vector space retrieval, probabilistic retrieval)
  - Index structures (inverted list, relational database)
  - Ranking of retrieved documents (RSV)

- We also look at a concrete implementation. Lucene is an open source project that provide reach text retrieval for many languages and environments.

2.2 Feature Extraction

- Normally, we do not search through documents with string operations. Rather, we extract characteristic features that describe the essence of the document in a concise way, and operate on these features only. In this chapter, we first look at lower level features that relate directly to the character sequence. Later on, we extract higher level features, for instance, classifiers, that describe the content with more abstract concepts.
- Feature extraction comprises of several steps which we subsequently analyze in more details:

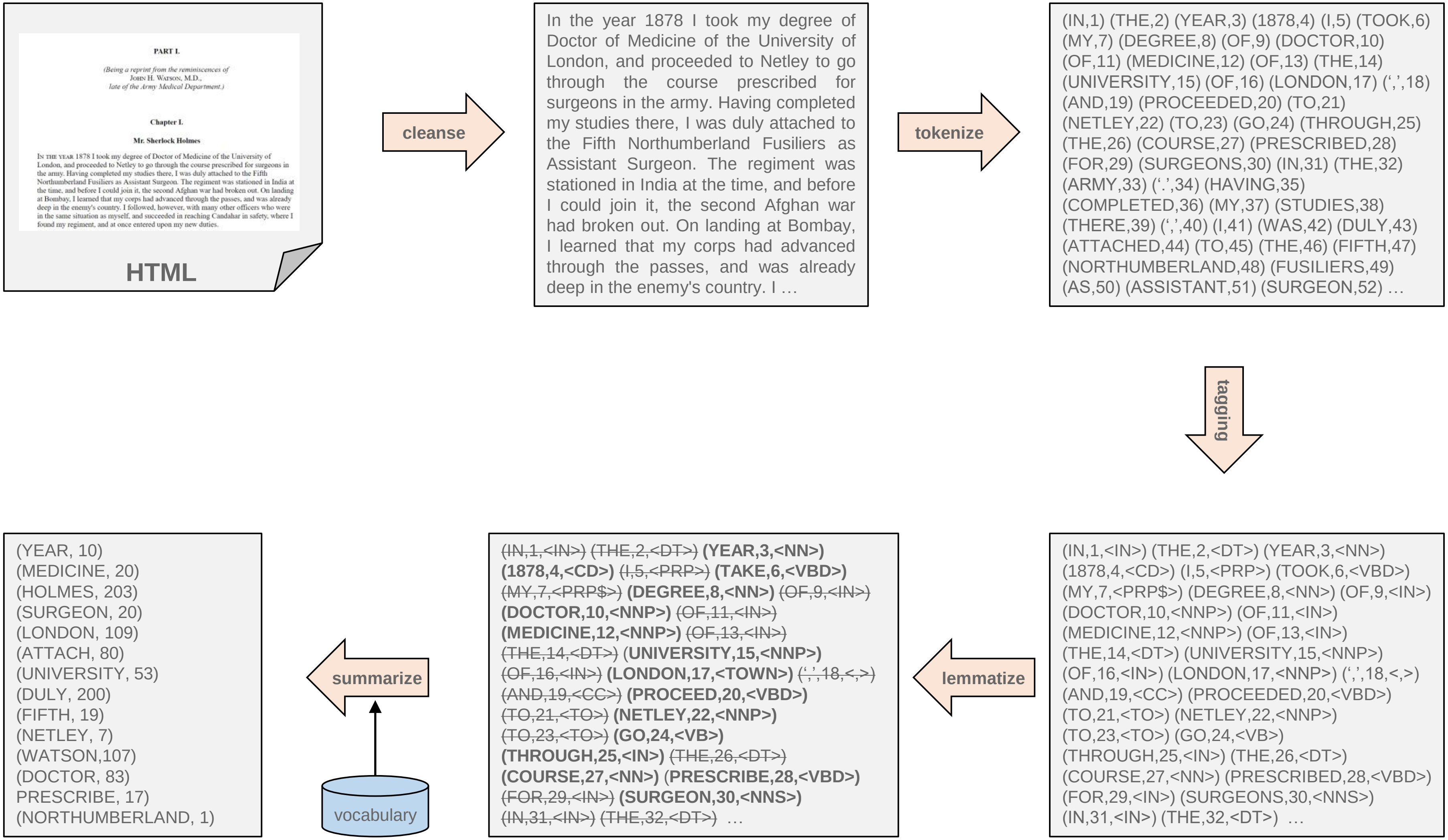
1. Cleanse document and reduce to sequence of characters
2. Create tokens from sequence
3. Tag token stream with additional information
4. Lemmatization, spell checking, and linguistic transformation
5. Summarize to feature vector (given a vocabulary)

- We are also looking into the python package NLTK which is a good starting point for advanced text processing. To get ready, ensure (as required for your Python environment):

```
sudo pip install -U nltk          # or pip3
sudo pip install -U numpy         # or pip3
python                            # or python3
   import nltk
   nltk.download()                # select: popular or all-nltk
```

- Apache OpenNLP is a good package for the Java world (also available through Lucene)

- Example of Feature Extraction



PART I.

(Being a reprint from the reminiscences of
John H. Watson, M.D.,
late of the Army Medical Department.)

Chapter I.

Mr. Sherlock Holmes

In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I followed, however, with many other officers who were in the same situation as myself, and succeeded in reaching Candahar in safety, where I found my regiment, and at once entered upon my new duties.

In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I …

(IN,1) (THE,2) (YEAR,3) (1878,4) (I,5) (TOOK,6) (MY,7) (DEGREE,8) (OF,9) (DOCTOR,10) (OF,11) (MEDICINE,12) (OF,13) (THE,14) (UNIVERSITY,15) (OF,16) (LONDON,17) (‘,’,18) (AND,19) (PROCEEDED,20) (TO,21) (NETLEY,22) (TO,23) (GO,24) (THROUGH,25) (THE,26) (COURSE,27) (PRESCRIBED,28) (FOR,29) (SURGEONS,30) (IN,31) (THE,32) (ARMY,33) (‘.’,34) (HAVING,35) (COMPLETED,36) (MY,37) (STUDIES,38) (THERE,39) (‘,’,40) (I,41) (WAS,42) (DULY,43) (ATTACHED,44) (TO,45) (THE,46) (FIFTH,47) (NORTHUMBERLAND,48) (FUSILIERS,49) (AS,50) (ASSISTANT,51) (SURGEON,52) …

(IN,1,<IN>) (THE,2,<DT>) (YEAR,3,<NN>) (1878,4,<CD>) (I,5,<PRP>) (TOOK,6,<VBD>) (MY,7,<PRP$>) (DEGREE,8,<NN>) (OF,9,<IN>) (DOCTOR,10,<NNP>) (OF,11,<IN>) (MEDICINE,12,<NNP>) (OF,13,<IN>) (THE,14,<DT>) (UNIVERSITY,15,<NNP>) (OF,16,<IN>) (LONDON,17,<NNP>) (‘,’,18,<,>) (AND,19,<CC>) (PROCEEDED,20,<VBD>) (TO,21,<TO>) (NETLEY,22,<NNP>) (TO,23,<TO>) (GO,24,<VB>) (THROUGH,25,<IN>) (THE,26,<DT>) (COURSE,27,<NN>) (PRESCRIBED,28,<VBD>) (FOR,29,<IN>) (SURGEONS,30,<NNS>) (IN,31,<IN>) (THE,32,<DT>) …

~~(IN,1,<IN>)~~ ~~(THE,2,<DT>)~~ **(YEAR,3,<NN>)** **(1878,4,<CD>)** ~~(I,5,<PRP>)~~ **(TAKE,6,<VBD>)** ~~(MY,7,<PRP$>)~~ **(DEGREE,8,<NN>)** ~~(OF,9,<IN>)~~ **(DOCTOR,10,<NNP>)** ~~(OF,11,<IN>)~~ **(MEDICINE,12,<NNP>)** ~~(OF,13,<IN>)~~ ~~(THE,14,<DT>)~~ (**UNIVERSITY,15,<NNP>)** ~~(OF,16,<IN>)~~ **(LONDON,17,<TOWN>)** ~~(‘,’,18,<,>)~~ ~~(AND,19,<CC>)~~ **(PROCEED,20,<VBD>)** ~~(TO,21,<TO>)~~ **(NETLEY,22,<NNP>)** ~~(TO,23,<TO>)~~ **(GO,24,<VB>)** **(THROUGH,25,<IN>)** ~~(THE,26,<DT>)~~ **(COURSE,27,<NN>)** (**PRESCRIBE,28,<VBD>)** ~~(FOR,29,<IN>)~~ **(SURGEON,30,<NNS>)** ~~(IN,31,<IN>)~~ ~~(THE,32,<DT>)~~ …

(YEAR, 10)
(MEDICINE, 20)
(HOLMES, 203)
(SURGEON, 20)
(LONDON, 109)
(ATTACH, 80)
(UNIVERSITY, 53)
(DULY, 200)
(FIFTH, 19)
(NETLEY, 7)
(WATSON,107)
(DOCTOR, 83)
PRESCRIBE, 17)
(NORTHUMBERLAND, 1)

2.2.1 Step 1: Cleanse Document (with the example of HTML)

- Text documents come in various formats like HTML, PDF, EPUB, or plain text. The initial step is to extract meta information and the sequence of characters that make up the text stream. This may include structural analysis of the document, encoding adjustments, and the identification of relevant information for the feature extraction. We do not want to index control sequences!
- Let us look at a simple example in HTML. The following snippet contains the rough structure of a web page. The first step is to identify which parts contain meaningful information. The header has rich meta information, the body contains the main text parts. Even though HTML is a well-defined standard, extracting information (so-called scraping) requires analysis of the data structure used for the documents. A web search engine simply considers everything.

```
<html>
   <head>
      <title> MMIR - SS01 </title>
      <meta name=„keywords“
        content=„multimedia, information,
               retrieval, course“>
   </head>

   <body>
      ...
      ...
   </body>
</html>
```

Header:
Contains meta-information about the document. We can use this information both for adding relevant features as well as cataloguing the document.

Body:
Contains the main content enriched with markups. The flow of the document is not always obvious and may look different on screen than in the file

- **Meta data:** the Web standards provide ways to define meta-information such as:
  - URI of page: (may contain concise key words)

    ```
    http://www-dbs.ethz.ch/~mmir/
    ```

  - Title of document: (concise summary of what to expect)

    ```
    <title>Multimedia Retrieval - Homepage</title>
    ```

  - Meta information in header section: (enriched information provided by author)

    ```
    <meta name=“keywords” content=“MMIR,information,retrieval,”>
    <meta name=“description” content=“This will change your life…”>
    ```

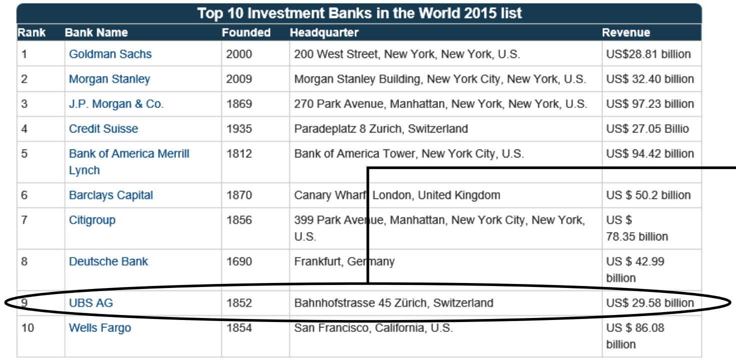
  The typical approach is to use the meta data for both the catalogue entry of the document and the text sequence. If we know the context of web pages, we can extract more accurate information.

- **Body Text:** the body subsumes all text blocks and tags them to control presentation. The flow on the page must not necessarily follow the order in the HTML file, but its typical a good enough approximation. Some of the tags provide useful additional information on the text pieces:
  - Headlines: `<h1>2. Information Retrieval </h1>`
  - Emphasized: `<b>Please read carefully!</b>`
    or `<i>Information Retrieval</i>`

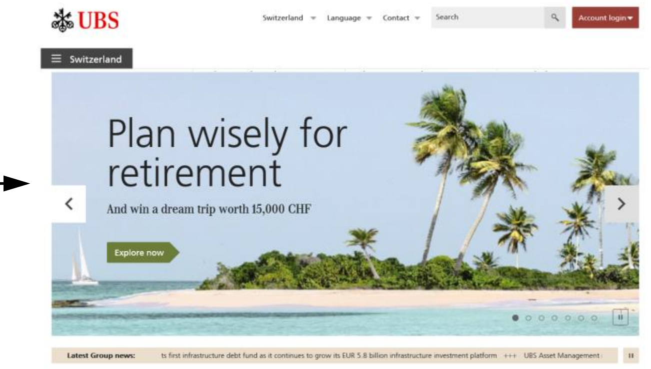
  A typical approach is to add meta information into the text stream to use the HTML tags during the processing steps. For instance, we could assign more weights to bold-faced terms.

- **Encoding:** most formats provide escape sequences or special characters, that need to be normalized. Furthermore, each document may use a different encoding which may lead to difficulties when searching for terms due to differences in representations
  - `&nbsp;` -> space, `&uuml;` -> ü
  - Transformation to Unicode, ASCII or other character set

- Web pages contain links. How do we handle them best? They describe relationships between documents and can add to the description of the current document. But more importantly, they also describe the referenced document. As authors of web pages keep link texts rather small, the set of keywords used by links is an good source for additional keywords in the referenced document.



Top 10 Investment Banks in the World 2015 list				
Rank	**Bank Name**	**Founded**	**Headquarter**	**Revenue**
1	Goldman Sachs	2000	200 West Street, New York, New York, U.S.	US$28.81 billion
2	Morgan Stanley	2009	Morgan Stanley Building, New York City, New York, U.S.	US$ 32.40 billion
3	J.P. Morgan & Co.	1869	270 Park Avenue, Manhattan, New York, New York, U.S.	US$ 97.23 billion
4	Credit Suisse	1935	Paradeplatz 8 Zurich, Switzerland	US$ 27.05 Billio
5	Bank of America Merrill Lynch	1812	Bank of America Tower, New York City, U.S.	US$ 94.42 billion
6	Barclays Capital	1870	Canary Wharf London, United Kingdom	US $ 50.2 billion
7	Citigroup	1856	399 Park Avenue, Manhattan, New York City, New York, U.S.	US $ 78.35 billion
8	Deutsche Bank	1690	Frankfurt, Germany	US $ 42.99 billion
9	UBS AG	1852	Bahnhofstrasse 45 Zürich, Switzerland	US$ 29.58 billion
10	Wells Fargo	1854	San Francisco, California, U.S.	US $ 86.08 billion



Plan wisely for retirement

And win a dream trip worth 15,000 CHF

Latest Group news: ts first infrastructure debt fund as it continues to grow its EUR 5.8 billion infrastructure investment platform +++ UBS Asset Management

- – Embedded objects (image, plug-ins):

```
        <IMG SRC=„img/MeAndMyCar.jpeg"
              ALT="picture of me in front of my car">
```

- – Links to external references:

```
        <a href=„http://anywhere.in.the.net/important.html“>
              read this important note </a>
```

- – **Approach:** Usually, the link text is associated with both the embedding and the linked document. However, we may weight keywords much higher when describing the referenced document with it. However, be aware of the effectiveness of such an approach, e.g., when considering click baits (promises much more than the referenced documents reveal) or navigational hints (“click here”, “back to main page”). We will look into this in the Web Retrieval chapter in more details.

2.2.2 Step 2: Create Tokens

- **Segmentation:** consider a book with several chapters, sections, paragraphs, and sentences. The goal of segmentation is to extract this meta structure from the text (often with the information provided by the previous step). While the broader segmentations (e.g., chapters) require control information from the document, sentence segmentation is possible on the text stream alone:
  - If we observe a ? or a !, a sentence ends (quite unambiguous, but this line is an exception)
  - The observation of a . (period) is rather ambiguous: it is not only used for sentence boundaries, but also in abbreviations, numbers, and ellipses that do not terminate a sentence
  - Some language specifics like ¿ in Spanish
  - Sentence-final particles that do not carry content information but add an effect to the sentence
    - Japanese: か *ka*: question. It turns a declarative sentence into a question.
      っけ *kke*: doubt. Used when one is unsure of something.
      な *na*: emotion. Used when one wants to express a personal feeling.
    - English: Don't do it, **man**. The blue one, **right**? The plate isn't broken, **is it**?
    - Spanish: Te gustan los libros, ¿**verdad**? Le toca pasar la aspiradora, ¿**no**?
  - A good heuristic works as follows (95% accuracy with English):

    1. If it is a ‘?’ or ‘!’, the sentence terminates
    2. If it is a ‘.’, then
        a. if the word before is a known abbreviation, then the sentence continues
        b. if the word afterwards starts with capital letter, then the sentence terminates

  - The approach in NLTK uses a trained method (Punkt) to determine sentence boundary.

- **Token Generation:** There are different ways to create tokens: a) Fragments of words, b) Words, and c) Phrases (also known a n-grams).
  - **Fragments of words:** an interesting approach in fuzzy retrieval is to break open words and extract sequences of characters (so-called k-grams) from the text stream. For example:

street	→ **str, tre, ree, eet**
streets	→ **str, tre, ree, eet, ets**
strets	→ **str, tre, ret, ets**

    An obvious advantage is that different inflections still appear similar at the fragment level. It also compensate for simple misspellings or bad recognition (OCR, speech analysis). Further, no language specific lemmatization is required afterwards. An early example included *EuroSpider* a search engine that used 3-grams to index OCR texts. However, while the technology was compelling, it has become superficial with the increased recognition and correction capabilities. In other retrieval scenarios, however, the method is still of interest. Music retrieval, DNA retrieval, and Protein Sequencing use fragments to model a sequence. In linguistic analysis, n-grams of words also play an important role for colocation analysis.
  - **Words**: using words as terms is the straightforward approach. But there as some subtle issues to deal with. For instance, how do you tokenize the following sequences?

Finland’s capital	→ Finland, Finlands, or Finland’s?
what’re, I’m, isn’t	→ What are, I am, is not?
l’ensemble	→ le ensemble?
San Francisco	→ one token or two?
m.p.h., PhD.	→ ??
$380.2, 20%	→ ??
Leuchtrakete	→ one word or composite word?

– **Words** (contd): In most languages, tokenization can use (space) separators between words. In Japanese and Chinese, words are not separated by spaces. For example:

莎拉波娃现在居住在美国东南部的佛罗里达。

莎拉波娃 现在 居住 在 美国 东南部 的 佛罗里达

Sharapova now lives in US southeastern Florida

In Japanese, texts can use different formats and alphabets mixed together.

- The conventional approach for tokenization is based on a regular expression to split words. One way to do so is as follows:

1. Match abbreviations with all upper case characters (e.g., U.S.A.)
2. Match sequences of word characters including hyphens (-) and apostrophes (‘)
3. Match numbers, currencies, percentage, and similar ($2.3, 20%, 0.345)
4. Match special characters and sequences (e.g., … ; “” ’’ () [])

- In addition, we may want to consider the special expressions/controls in the environment like hashtags (#blowsyourmind), user references (@thebigone), emoticons (☺), or control sequences in the format (e.g., wiki).
- NLTK uses the Treebank tokenizer and the Punkt tokenizer depending on the language. There are a few simpler methods that split sequences on whitespaces or regular expression.
- For Japanese and Chinese, we identify token boundaries with longest matches that lead a known word from the dictionary. This approach would not work in other languages if we would omit spaces.

- **Phrases:** we have seen some examples, where it seems more appropriate to consider several words as a singular term (e.g., New York, San Francisco, Sherlock Holmes). In other examples, the combinations of two or more words can change or add to the meaning beyond the words. Examples include express lane, crystal clear, middle management, thai food, Prime Minister, and other compounds. To capture them, we can extract so-called n-grams from the text stream:

  1. Extract the base terms (as discussed before)
  2. Iterate through the term sequence
     - Add 2-grams, 3-grams, …, n-grams over subsequent terms at a given position

  However, this leads to many meaningless compounds such as “the house”, “I am”, “we are”, or “it is” which are clearly not interesting to us. More over, we generate thousands of new term groups that are just accidentally together (like “meaningless compounds” or “better control” in this paragraph). To better control the selection of n-grams, various methods have been proposed. We consider here only two simple and intuitive measures:

  – A simple approach is to reject n-grams that contain at least one so-called stop word. A stop word is a linguistic element that bears little information in itself. Examples include: a, the, I, me, your, by, at, for, not, … Although very simple, this already eliminates vast amounts of useless n-grams.

  – **Pointwise Mutual Information (PMI).** For simplicity, we consider only the case of 2-grams but generalization to n-grams is straightforward. The general idea is that the 2-gram is interesting only if it occurs more frequently than the individual distributions of the two terms would suggest (and assuming they are independent). To this end, we can compute the Pointwise Mutual Information pmi for two terms t_1 and t$_2$ as follows:

$$pmi(t_1,t_2) = \log\frac{p(t_1,t_2)}{p(t_1)\cdot p(t_2)} = \log\frac{p(t_1|t_2)}{p(t_1)} = \log\frac{p(t_2|t_1)}{p(t_2)} = \log p(t_1,t_2) - \log p(t_1) - \log p(t_2)$$

- **Pointwise Mutual Information** (contd): Let $p(t_j)$ be the probability that we observe the term t_j in the text. We compute this probability with a maximum likelihood approach. Let M be the number of different terms in the collection and $tf(t_j)$ be the so-called **term frequency** of term t_j with a total of N occurrences of terms in the text. We then obtain $p(t_j)$ as follows:

  $$p(t_j) = \frac{tf(t_j)}{N} \qquad \forall j: 1 \leq j \leq M$$

  similarly: $$p(t_1, t_2) = \frac{tf(t_1, t_2)}{N}$$

  Now, assume we have two terms t_1 and t_2. If they are independent from each other, then the probability $p(t_1, t_2)$ of their co-occurrence is the product of their individual probabilities $p(t_j)$ and the pmi becomes 0. If t_2 always follows t_1, then $p(t_2|t_1) = 1$ and the pmi is positive and large. If t_2 never follows t_1, then $p(t_2|t_1) = 0$ and $pmi = -\infty$. Hence, we keep 2-grams if their pmi is positive and large, and otherwise dismiss them. In addition, we dismiss infrequent 2-grams with $tf(t_1, t_2) < threshold$ to avoid accidental co-occurrences with high pmi (seldom words):

Bigram	$tf(t_1)$	$tf(t_2)$	$tf(t_1, t_2)$	$pmi(t_1, t_2)$
salt lake	11	10	10	11.94
halliday private	5	12	5	11.81
scotland yard	8	9	6	11.81
lake city	10	23	9	10.72
private hotel	12	14	6	10.59
baker street	6	29	6	10.54
brixton road	15	28	13	10.38
jefferson hope	37	56	34	9.47
joseph stangerson	13	47	10	9.46
enoch drebber	8	62	8	9.44
old farmer	39	9	5	9.26
john rance	39	10	5	9.11
john ferrier	39	62	29	9.01
sherlock holmes	52	98	52	8.78

2.2.3 Step 3: Tagging of Tokens

- A simple form of tagging is to add position information to the tokens. Usually, this is already done at token generation time (term position in stream).
- For natural language processing, tagging associates a linguistic or lexical category to the term. With **Part of Speech (POS)**, we label terms as nouns, verbs, adjectives, and so on. Based on this information, we can construct tree banks to define the syntactic and semantic structure of a sentence. Tree banks have revolutionized computational linguistic in the 1990s with “The Penn Treebank” as first large-scale empirical data set. It defines the following tags:

Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Proper nouns are specific people, places, things.

Tag	Description
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Wh-determiner
WP	Wh-pronoun
WP$	Possessive wh-pronoun
WRB	Wh-adverb

with NLTK, use `nltk.help.upenn_tagset()`

WH-words are: where, what, which, when, …

- NLTK also provides a simpler variant with the universal POS tagset. It is based on the same (machine learning) approach as Penn Treebank but maps tags to a smaller/simpler set. Here is an example together with the number of occurrences in “A Study in Scarlet”:

Tag	Description	Freq	Examples
ADJ	adjective	2812	new, good, high, special, big, local
ADP	adposition	5572	on, of, at, with, by, into, under
ADV	adverb	2607	really, already, still, early, now
CONJ	conjunction	1711	and, or, but, if, while, although
DET	determiner, article	5307	the, a, some, most, every, no, which
NOUN	noun	9358	year, home, costs, time, Africa
NUM	numeral	354	twenty-four, fourth, 1991, 14:24
PRT	particle	1535	at, on, out, over per, that, up, with
PRON	pronoun	5705	he, their, her, its, my, I, us
VERB	verb	8930	is, say, told, given, playing, would
.	punctuation marks	7713	. , ; !
X	other	36	ersatz, esprit, dunno, gr8, univeristy

POS tags are the basis for natural language processing (NLP). They are used to define a parse tree which allows the extraction of context and the transformation of sentences. **Named entities** is one such transformation. Based on the initial POS tagging and with the help of a entity database, individual tokens or groups of tokens are collapsed to a single named entity.

Chunking is the more generic technique. We can define a simple grammar which is used to construct **non-overlapping phrases (NP)**. For example, the grammar “`NP: {<DT>?<JJ>*<NN>}`“ collapses a sequence of article, adjectives, and noun into a new group.

2.2.4 Step 4: Lemmatization and Linguistic Transformation

- Lemmatization and linguistic transformation are necessary to match query terms with document terms even if they use different inflections or spellings (colour vs. color). Depending on the scenario, one or several of the following methods can be applied.
- A very common step is **stemming**. In most languages, words appear in many different inflected forms depending on time, case, or gender. Examples:
  - English: go, goes, went, going, house, houses, master, master’s
  - German: gehen, gehst, ging, gegangen, Haus, Häuser, Meister, Meisters

  As we see from the example, the inflected forms vary greatly but essentially do mean the same. The idea of stemming is to reduce the term to a common stem and use this stem to describe the context. In many languages, like German, stemming is challenging due to its many irregular forms and the use of strong inflection (gehen → ging). In addition, some languages allow the construction of “new terms” through compound techniques which may lead to arbitrarily long words:
  - German (law in Mecklenburg-Vorpommern, 1999-2013): Rinderkennzeichnungs- und Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz. Literally ‘cattle marking and beef labeling supervision duties delegation law’
  - Finnish: *atomiydinenergiareaktorigeneraattorilauhduttajaturbiiniratasvaihde.* Literally 'atomic nuclear energy reactor generator condenser turbine cogwheel stage’

  In many cases, we want to decompose the compounds to increase chances to match against query terms. Otherwise, we may never find that German cattle law with a query like “Rind Kennzeichnung”. On the other side, breaking a compound may mislead from the true meaning
  - German: Gartenhaus → Garten, Haus (ok, not too far away from the true meaning)
  - German: Wolkenkratzer → Wolke, Kratzer (no, this is completely wrong)

- For English, the **Porter Algorithm** determines a near-stem of words that is not linguistic correct but in most cases, words with the same linguistic stem are reduced to same near-stem. The algorithm is very efficient and several extensions have been proposed in the past. We consider here the original version of Martin Porter from 1980:
  - Porter defines character `v` as a „vocal" if
    - it is an **A, E, I, O, U**
    - it is a **Y** and the preceding character is not a „vocal" (e.g. R**Y**, B**Y**)
  - All other characters are consonants (`c`)
  - Let `C` be a sequence of consonants, and let `V` be a sequence of vocals
  - Each word follows the following pattern:
    - $[C](VC)^{m}[V]$
    - `m` is the measure of the word
  - further:
    - `*o`: stem ends with cvc; second consonant must not be W, X or Y (-WIL, -HOP)
    - `*d`: stem with double consonant (-TT, -SS)
    - `*v*`: stem contains a vocal
  - The following rules define mappings for words with the help of the forms introduced above. `m` is used to avoid overstemming of short words.

Source: Porter, M.F.: *An Algorithm for Suffix Stripping*. Program, Vol. 14, No. 3, 1980

– Porter algorithm - extracts (1)

	Rule		Examples	
	Step 1			
a)	SSES	-> SS	caresses	-> caress
	IES	-> I	ponies	-> poni
	SS	-> SS	caress	-> caress
	S	->	cats	-> cat
b)	(m>0) EED	->EE	feed	-> feed
	(*v*) ED	->	plastered	-> plaster
	(*v*) ING	->	motoring	-> motor
	... (further rules)			
	Step 2			
	(m>0) ATIONAL	-> ATE	relational	-> relate
	(m>0) TIONAL	-> TION	conditional	-> condition
	(m>0) ENCI	-> ENCE	valenci	-> valence
	(m>0) IZER	-> IZE	digitizer	-> digitize
	... (further rules)			

- Porter algorithm - extracts (2)

	Rule		Examples	
	Step 3			
	`(m>0) ICATE`	`-> IC`	triplicate	-> triplic
	`(m>0) ATIVE`	`->`	formative	-> form
	`(m>0) ALIZE`	`-> AL`	formalize	-> formal
	... (further rules)			
	Step 4			
	`(m>1) and (*S or *T)ION`	`->`	adoption	-> adopt
	`(m>1) OU`	`->`	homologou	-> homolog
	`(m>1) ISM`	`->`	platonism	-> platon
	... (further rules)			
	Step 5			
a)	`(m>1) E`	`->`	rate	-> rate
	`(m=1) and (not *o)E`	`->`	cease	-> ceas
b)	`(m>1 and *d and *L)`	`-> single letter`	controll	-> control

- There are several variants and extensions of the Porter Algorithm. **Lancaster** uses a more aggressive stemming algorithm that can result in almost obfuscated stems but at increased performance. **Snowball** is a set of rule based stemmers for many languages. An interesting aspect is the domain specific language to define stemmers, and compilers to generate code in many computer languages.
- In contrast to the rule based stemmers, a dictionary based stemmer reduces terms to a linguistic correct stem. This comes at additional stemming costs and the need to maintain a dictionary. The EuroWordNet initiative develops a semantic dictionary for many of the European languages. Next to words, the dictionary also contain all inflected forms, a simplified rule-based stemmer for regular inflections, and semantic relations between words (so-called ontologies).
  - Examples of such dictionaries / ontologies:
    - EuroWordNet: `http://www.illc.uva.nl/EuroWordNet/`
    - GermaNet: `http://www.sfs.uni-tuebingen.de/lsd/`
    - WordNet: `http://wordnet.princeton.edu/`
  - We consider in the following the English version of WordNet with its stemmer Morphy. It consists of three parts
    - a simple rule-based stemmer for regular inflections (-ing, -ed, …)
    - an exception list for irregular inflections
    - a dictionary of all possible stems of the language

- The rule-based approach is quite similar to the Porter rules but they only apply to certain word types (noun, verb, adjective).
- The stemming works as follows:

1. Search the current term in the dictionary. If found, return the term as its own stem (no stemming required)
2. Search the current term in the exception lists. If found, return the associated linguistic stem (see table below)
3. Try all rules as per the table on the right. Replace the suffix with the ending (we may not know the word type, so we try all of them)
    a. If a rule matches, search in the indicated dictionary for the reduced stem. If found, return it as the stem
    b. If several rules succeed, choose the more likely stem
    Example: axes → axis, axe
4. If no stem is found, return the term as its own stem

Type	**Suffix**	**Ending**
NOUN	s	
NOUN	ses	s
NOUN	xes	x
NOUN	zes	z
NOUN	ches	ch
NOUN	shes	sh
NOUN	men	man
NOUN	ies	y
VERB	s	
VERB	ies	y
VERB	es	e
VERB	es	
VERB	ed	e
VERB	ed	
VERB	ing	e
VERB	ing	
ADJ	er	
ADJ	est	
ADJ	er	e
ADJ	est	e

adj.exc (1500):

...	
stagiest	stagy
stalkier	stalky
stalkiest	stalky
stapler	stapler
starchier	starchy
starchiest	starchy
starer	starer
starest	starest
starrier	starry
starriest	starry
statelier	stately
stateliest	stately
...	

verb.exc (2400):

...	
ate	eat
atrophied	atrophy
averred	aver
averring	aver
awoke	awake
awoken	awake
babied	baby
baby-sat	baby-sit
baby-sitting	baby-sit
back-pedalled	back-pedal
back-pedalling	back-pedal
backbit	backbite
...	

noun.exc (2000):

...	
neuromata	neuroma
neuroptera	neuropteron
neuroses	neurosis
nevi	nevus
nibelungen	nibelung
nidi	nidus
nielli	niello
nilgai	nilgai
nimbi	nimbus
nimbostrati	nimbostratus
noctilucae	noctiluca
...	

- NLTK supports Porter, Lancaster, Snowball and WordNet stemmers. An example of all stemmers at works yields the following table for comparison. Note that the Morphy implementation in NLTK requires a hint for the word type, otherwise it considers the term as a noun.

Term	Porter Stem	Lancaster Stem	Snowball Stem	WordNet Stem
took	took	took	took	take
degree	degre	degr	degre	degree
doctor	doctor	doct	doctor	doctor
medicine	medicin	medicin	medicin	medicine
university	univers	univers	univers	university
proceeded	proceed	process	proceed	proceed
course	cours	cours	cours	course
surgeons	surgeon	surgeon	surgeon	surgeon
army	armi	army	armi	army
completed	complet	complet	complet	complete
studies	studi	study	studi	study
there	there	ther	there	there
was	wa	was	was	be
duly	duli	duly	duli	duly
fifth	fifth	fif	fifth	fifth
fusiliers	fusili	fusy	fusili	fusiliers
assistant	assist	assist	assist	assistant
regiment	regiment	regy	regiment	regiment
stationed	station	stat	station	station
time	time	tim	time	time
afghan	afghan	afgh	afghan	afghan
had	had	had	had	have
broken	broken	brok	broken	break

- When analyzing text or parsing a user query, we will come across **homonyms** (equal terms but different semantics) and **synonyms** (different terms but equal semantics). Homonyms may require additional annotations from the context to extract the proper meaning. Synonyms are useful to expand a user query if the original search is not (that) successful. Examples:
  - **Homonyms** (equal terms but different semantics):
    - bank (shore vs. financial institute)
  - **Synonyms** (different terms but equal semantics):
    - walk, go, pace, run, sprint

  WordNet groups English words into so-called synsets or synonym sets and provides short definitions of their usage. Furthermore, it contains further relations among these synsets:
  - **Hypernyms** (umbrella term) / **Hyponym** (species)
    - **Animal** ← dog, cat, bird, ...
  - **Holonyms** (is part of) / **Meronyms** (has parts)
    - **door** ← lock

  These relationships define a knowledge structure. The hypernym/hyponym relationship defines a hierarchy with synsets at each level and the unique beginner synset “entity”. We can use this structure to derive further information or context data for our annotations. For instance, if we find the term horse, we can try to derive whether the text is about an animal or about a chess figure.
  - NLTK provides the corpus nltk.corpus.wordnet which provides access to the WordNet knowledge structure. You can also browse through the structure online.
- **Spell checking:** for user queries, we often use spell checkers to fix simple misspellings or to suggest corrected versions of the terms. Most systems provide a fuzzy search which automatically looks for similar terms and adds them to the query if necessary (see Lucene later on)

2.2.5 Step 5: Summarize to Feature Vector

- Before we can create a feature vector, we first must define the vocabulary and decide how to statistically summarize the term information.
- **Vocabulary:** how many different terms does a collection of documents contain? Church and Gale gave a very good and rough estimator: the number of distinct terms is about the square root of the number of tokens in the entire collection. But not all of these terms are equally important for the retrieval task. So how can we find the most important ones?
  - We usually normalize terms before we add them to the vocabulary (but this is not necessary). As discussed in the previous section, we may end up with near-stems or real stems of the words. Normalization not only reduces the size of vocabulary but it also merges different terms with (mostly) the same meaning. For instance:
    - we can consider the set {cat, cats, cat’s, cats’} as 4 individual terms or as a single term
    - we can treat a synset as one term or each constituent of the synset as an individual term
  - Regardless of the chosen method to extract and normalize terms, we want to eliminate terms that do not help us much with describing the content of the document. For instance, the term ‘it’ is used in almost every English text and bears little information about the content. So we may want to ignore these so-called **stop words**; here some examples for English:

> i me my myself we our ours ourselves you your yours yourself yourselves he him his himself she her hers herself it its itself they them their theirs themselves what which who whom this that these those am is are was were be been being have has had having do does did doing a an the and but if or because as until while of at by for with about against between into through during before after above below to from up down in out on off over under again further then once here there when where why how all any both each few more most other some such no nor not only own same so than too very s t can will just don should now d ll m o re ve y ain aren couldn didn doesn hadn hasn haven isn ma mightn mustn needn shan shouldn wasn weren won wouldn

- Stop word elimination is very common but bears some risks if not done carefully. In the example before, we stated that “it” is not meaningful to distinguish English texts. But consider this:
  - Stephen King wrote a book “It” – We never will find this book if we eliminate ‘it’ as a stop word
  - If we write IT we actually mean information technology – even though it looks like our ‘it’, the big IT is a homonym with a very distinct meaning
  - What do you get if you search the web for ‘it’?
- The other extreme case are seldom terms (or bigrams, n-grams) that only appear once in the entire collection. This multimedia retrieval course is the only one containing the bigram **endoplasmic reticulum**. Is it worth to index this bigram? Is any student ever going to search for this in a computer science collection? If this is unlikely, why bother with such terms.
  - We already considered the pmi earlier when we extracted n-grams from the text. pmi is a simple measure to reduce the numbers of n-grams that we want to consider. Without such a control, we would end up with excessive numbers of terms. According to the Oxford English Dictionary, there are about 170’000 currently used words in English. With bigrams, the potential number is in the billions, and with n-grams (and large corpuses) we may obtain trillions of combinations (upper bound by the number of tokens in the collection). Google’s n-gram viewer has 1 trillion tokens but “only” 13 million n-grams. Clearly, rare combinations were taken off the vocabulary. So filtering rare terms is an important step.
- A final issue are spelling mistakes. Britney, Britni, Bridney, Britnei all appear similar but are different terms for our retrieval system. Misspellings not only blow up our vocabulary (consider all spelling mistakes ever done by all people!), but they also make it impossible to retrieve the content by the correct spelling. On the other side, all of the names given before do also exist (maybe in some cases the parents misspelled the name on the form)

– A pragmatic approach to control vocabulary size is based on **Zipf's law**. Let N be the total number of term occurrences (tokens) in the collection and M be the number of distinct terms in the vocabulary. We already used the term frequency $tf(t)$ to denote the number of occurrences of term t. Now, let us order all terms by decreasing term frequencies and assign $rank(t)$ to term t based on that order. The central theorem of Zip's law is that the probability p_r of randomly selecting the term t with $rank(t) = r$ from the collection is

$p_r = \frac{c}{r} = \frac{tf(t)}{N}$ for the term t with $rank(t) = r$. c is a constant depending only on M

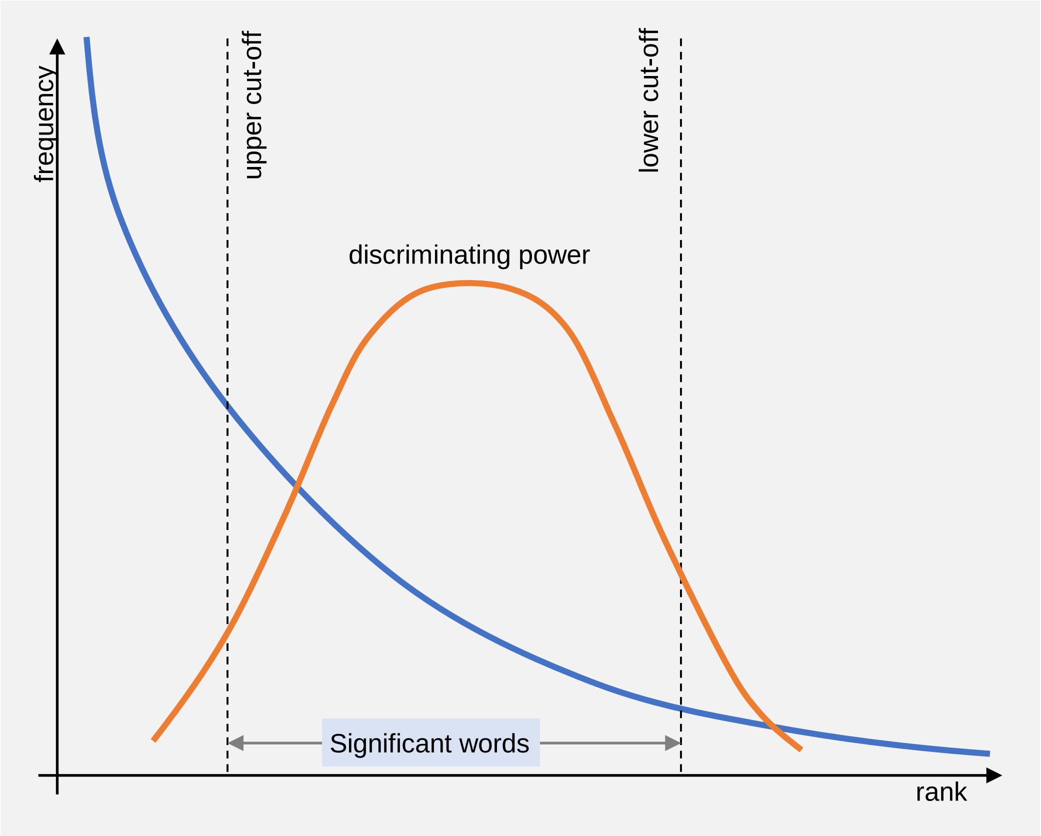
In other words, we always get the same constant value $c \cdot N$ if we multiply the rank of a term with its term frequency. Or we can estimate the rank of a term t as: $rank(t) = c \cdot \frac{N}{tf(t)}$. We can easily compute c as a function of M as follows:

$$1 = \sum_{r=1}^{M} p_r = \sum_{r=1}^{M} \frac{c}{r} = c \cdot \sum_{r=1}^{M} \frac{1}{r} \quad \rightarrow \quad c = \frac{1}{\sum_{r=1}^{M} \frac{1}{r}} \approx \frac{1}{0.5772 + \ln M}$$

With this we get a simple lookup table for c given the number M of distinct terms:

M	5'000	10'000	50'000	100'000
c	0.11	0.10	0.09	0.08

– The right hand figure shows the Zipf distribution. As discussed, the most frequent words (above the upper cut-off line) bear little meaning as they occur in almost every text. The least frequent words (below the lower cut-off) appear too seldom to be used in queries and only discriminate a few documents. The range of significant words lies in between the lower and upper cut-off.



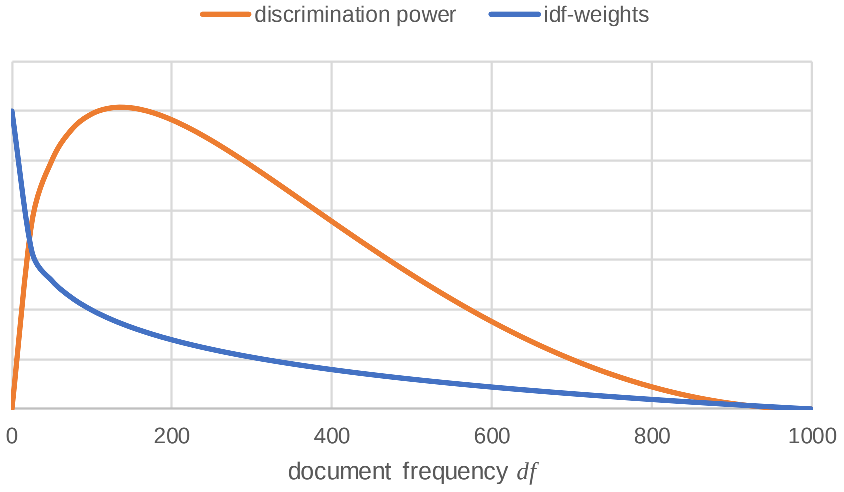
– Originally, the idea was to define threshold and eliminate the words outside the indicated range. This would save memory and speed up search. This has become irrelevant.

– Today, the typical approach is to eliminate only stop words from a short well-maintained list, or to keep even all terms as the additional (storage) overhead is minimal. In the latter case, we use weights depending on the frequency of terms. Or more generically, based on the discriminative power the terms possess. With this weights, we can express how well a term can distinguish between relevant and non-relevant documents. The figure above indicates that power of discrimination with the red plot. Note that even though the very rare terms are directly pointing to the relevant documents, they are also rarely used in queries and, hence, their expected discrimination power is low. The best terms are those, that divide documents clearly (contain term, do not contain term) and are frequently used in queries.

- **Discrimination power:** in vector space retrieval, we use the so-called **inverse document frequency** to define weights on terms that correspond directly to their discrimination power. Instead of counting the total number of occurrences as in the term frequency $tf(t)$, the document frequency $df(t)$ counts in how many documents the term t appears at least once. Let N be the number of documents in the collection. The inverse document frequency $idf(t)$ is then given as (note that there are many similar definitions):

$$idf(t) = \log \frac{N+1}{df(t)+1} = \log(N+1) - \log(df(t)+1)$$

The inverse document frequency denotes a weight on the term used both in the document description as in the query description. We can estimate the discrimination power of a term t by multiplying the squared $idf(t)$-value with the probability that the term occurs in the query, or its expected contribution to the ranking (=discrimination power).The figure below shows idf-weights (blue) and discrimination power (red) as a function of the document frequency df and with $N = 1000$ documents.



- Terms with low document frequencies (on the left side) received the highest idf-weights but as they also seldom appear in queries, their discrimination power is low
- On the right side, the terms with high document frequency have both low weights and discrimination power.
- The terms around $df = 100 = 0.1 \cdot N$ have the highest discrimination power.

- The **discrimination method** provides and alternative to the idf-weights. In essence, we want to measure how much a term is able to discriminate the document collection, or from a different angle: if we remove the term from the collection, how much more similar do the documents become without that term. Let $0 \le sim(D_i, D_j) \le 1$ denote the similarity between two documents D_i and D_j where 0 means the documents are totally dissimilar and 1 means they are identical.
  - In a collection with N documents, compute the centroid document C as the document that contains all M terms with mean frequency over the collection. If t$f(D_i, t_j)$ is the term frequency of term t_j in document D_i, then

$$tf(C, t_j) = \frac{1}{N} \cdot \sum_{i=1}^{N} tf(D_i, t_j) \quad \text{for } \forall j: 1 \le j \le M$$

  - We define the density of the collection as the sum of all similarities between documents and their centroid C:

$$Q = \sum_{i=1}^{N} sim(D_i, C)$$

  - Now assume we remove the term t from the collection. We can compute the density Q_t for this modified collection and then define the discrimination power of term t as:

$$dp(t) = Q_t - Q$$

  - If the discrimination value is large, Q_t is larger than Q. Hence, if we remove the term t from the collection, similarities to the centroid become larger. If we add the term again, documents become more distinct from their centroid. In other words, the term t differentiates the collection and is hence a significant term. On the other side, if $dp(t)$ is negative, we conclude that Q is larger than Q_t. That means if we remove the term from the collection, documents become more distinct from the centroid. If we add the term again, the documents become more similar to the centroid. In other words, the term is likely “spamming” the collection and has a (very) negative impact on describing the documents. For example, if we add the term “hello” a 1’000 times to each document, they obviously become more similar to each other (and the centroid). Hence, terms with very small $dp(t)$ are not significant (or even harmful) to describe the collection.
  - We can now select the most useful terms by ordering them by their decreasing $dp(t)$-values and cut-off the list if the discrimination value falls below some threshold value.
- Once the vocabulary is fixed, we can describe documents D_i by a feature value d_i. The **set-of words** model is a simple representation that only considers whether a term is present and disregards order of terms, number of occurrences, and proximity between terms. The most simple representation is simply the set of terms appearing at least once, or a binary feature vector where dimension j denotes the presence ($=1$) or absence ($=0$) of term t_j.

$$d_{i,j} \in \{0,1\}^M, \qquad d_{i,j} = \begin{cases} 1 & tf\left(D_i, t_j\right) > 0 \\ 0 & tf\left(D_i, t_j\right) = 0 \end{cases} \qquad \text{or} \quad d_i = \left\{t_j \mid tf\left(D_i, t_j\right) > 0\right\}$$

The **bag-of-of words** model is the more common representation and differs from the set-of-words by keeping multiplicity of terms. The representation is a feature vector over term frequencies

$$d_{i,j} \in \mathbb{N}^M, \qquad d_{i,j} = tf\left(D_i, t_j\right)$$

2.3 Text Retrieval Models

- In the following sections we consider different retrieval models and discuss their advantages and disadvantages. We only touch the essential method while there many more extensions in the literature. We will use the following notations in this chapter:

Notation	Value Range	Description
$\mathbb{D}$	$\{D_1, \ldots, D_N\}$	Collection of N documents
D_i		Representation of a document with $1 \leq i \leq N$
$\mathbb{T}$	$\{t_1, \ldots, t_M\}$	Collection of M terms
t_j		Representation of a term with $1 \leq j \leq M$
$\boldsymbol{d}_i$	$\{0,1\}^M$, $\mathbb{N}^M$, or $\mathbb{R}^M$	Feature description of document D_i with the j-the dimension describing document with regard to term t_j
$\mathbf{A}$	$\{0,1\}^{M \times N}$, $\mathbb{N}^{M \times N}$, or $\mathbb{R}^{M \times N}$	Term-document matrix with $a_{j,i} = tf(D_i, t_j)$, that is rows denote terms and columns denote documents. For instance, the i-th column is $a_{:,i} = \boldsymbol{d}_i$.
$tf(D_i, t_j)$	$\mathbb{N}$	Term frequency of term t_j in document D_i, i.e., number of occurrences of term t_j in document D_i
$df(t_j)$	$\mathbb{N}$	Document frequency of term t_j in the collection $\mathbb{D}$, i.e., number of documents in $\mathbb{D}$ that contain term t_j at least once
$idf(t_j)$	$\mathbb{R}$	Inverse document frequency of term t_j given by $idf(t_j) = \log(N+1) - \log(df(t_j) + 1)$
Q		Representation of a query
$\boldsymbol{q}$	$\{0,1\}^M$, $\mathbb{N}^M$, or $\mathbb{R}^M$	Feature description of query Q with the j-the dimension describing query with regard to term t_j
$sim(Q, D_i)$	[0,1]	Similarity between query Q and document D_i. 0 means dissimilar, 1 means identical

2.3.1 Standard Boolean Model

- The standard Boolean model is the classical text retrieval method introduced in the 1970s. Given the limited capabilities of computing at this time, it was important that we can answer queries by considering only the current data set (tape drives were sequential). Even though more advanced methods were developed, it is still used by many methods and works fairly well.
- As the names suggests, the model operates on Boolean logic over sets of terms. Documents are represented by sets of words, and queries come from the following grammar:

• $Q = t$	Term t must be present
• $Q = \neg t$	Term t must not be present
• $Q = Q_1 \vee Q_2$	Sub-query q_1 or sub-query q_2 fullfilled
• $Q = Q_1 \wedge Q_2$	Both sub-query q_1 and q_2 fullfilled

- To evaluate such queries, we can transform them into their disjunctive normal form

$$Q = \left(\tau_{1,1} \ \wedge \cdots \wedge \ \tau_{1,K_1}\right) \vee \cdots \vee \ \left(\tau_{L,1} \ \wedge \cdots \wedge \ \tau_{L,K_L}\right) = \bigvee_{l=1}^{L} \left(\bigwedge_{k=1}^{K_l} \tau_{l,k} \right)$$

with $\tau_{l,k} = t_{j(l,k)}$ or $\tau_{l,k} = \neg t_{j(l,k)}$

- For each atomic part $\tau_{l,k}$, we can compute the set $\mathbb{S}_{l,k}$ of documents that contain or do not contain the term:

$$\mathbb{S}_{l,k} = \begin{cases} \left\{D_i \mid tf\left(D_i, t_{j(l,k)}\right) = 1\right\} & \text{if } \tau_{l,k} = t_{j(l,k)} \\ \left\{D_i \mid tf\left(D_i, t_{j(l,k)}\right) = 0\right\} & \text{if } \tau_{l,k} = \neg t_{j(l,k)} \end{cases}$$

- The final result $\mathbb{Q}$ is then a combination of intersections and unions over the sets derived from the atomic parts

$$\mathbb{Q} = \bigcup_{l=1}^{L} \bigcap_{k=1}^{K_l} \mathbb{S}_{l,k} = \bigcup_{l=1}^{L} \bigcap_{k=1}^{K_l} \begin{cases} \{D_i \mid tf(D_i, t_{j(l,k)}) = 1\} & \text{if } \tau_{l,k} = t_{j(l,k)} \\ \{D_i \mid tf(D_i, t_{j(l,k)}) = 0\} & \text{if } \tau_{l,k} = \neg t_{j(l,k)} \end{cases}$$

- **Advantages:** simple model with a clean description of query semantics. Very simple to implement and intuitive for users. Even though the definition of query evaluation is based on sets, we will see later in this chapter that the inverted lists provides a very efficient way to compute the inner intersections of the evaluation (with some restrictions on query structure). The Boolean expression provides an accurate way to define what relevance means.
- **Disadvantages:** no (intuitive) control over the size of retrieved documents and a user may get either too few or too many results. For larger result sets, the lack of ranking requires the user to browse through the documents to find the best match. Although the query language is simple, users may find it hard to express a complex information need as a combination of ANDs and ORs. All terms are treated equally, hence, stop words contribute equally to the result as the most significant terms. Retrieval quality is ok but other methods (with equal computational complexity) achieve much better results.

2.3.2 Extended Boolean Model

- The lack of ranking is a huge handicap of the standard Boolean model. The extended versions of the Boolean model overcome this drawback: we consider term weights, use a bag of words model, and apply partial matching similar to the vector space retrieval model. The algebra is still Boolean but evaluations return a similarity value rather than a 0/1-view. There are several variants but all following a similar concept
- A document D_i is represented as a vector $\boldsymbol{d}_i$ with normalized term frequencies:

$$d_{i,j} = \min\left(1, \frac{tf\left(D_i, t_j\right) \cdot idf\left(t_j\right)}{\alpha}\right) \quad \forall j: 1 \leq j \leq M \quad \text{with } \alpha = \max\left(tf\left(D_i, t_j\right) \cdot idf\left(t_j\right)\right) \quad \text{(or some other value)}$$

Other methods to normalization are possible (like the discrimination value we have seen previously). A query Q follows the same structure as in the standard Boolean model, hence:

$$Q = \left(\tau_{1,1} \ \wedge \cdots \wedge \ \tau_{1,K_1}\right) \vee \cdots \vee \ \left(\tau_{L,1} \ \wedge \cdots \wedge \ \tau_{L,K_L}\right) = \bigvee_{l=1}^{L}\left(\bigwedge_{k=1}^{K_l} \tau_{l,k}\right)$$

with $\tau_{l,k} = t_{j(l,k)}$ or $\tau_{l,k} = \neg t_{j(l,k)}$

- For each atomic part $\tau_{l,k}$, we can compute the similarity value $sim\left(Q = \tau_{l,k}, D_i\right)$ for a document D_i:

$$sim(Q = \tau_{l,k}, D_i) = \begin{cases} d_{i,j(l,k)} & \text{if } \tau_{l,k} = t_{j(l,k)} \\ 1 - d_{i,j(l,k)} & \text{if } \tau_{l,k} = \neg t_{j(l,k)} \end{cases}$$

- There are several variants that calculate the AND and OR operators.
  - **Fuzzy Algebraic:** (only works for two operands)

$$sim(Q_1 \wedge Q_2, D_i) = sim(Q_1, D_i) \cdot sim(Q_2, D_i)$$
$$sim(Q_1 \vee Q_2, D_i) = sim(Q_1, D_i) + sim(Q_2, D_i) - sim(Q_1, D_i) \cdot sim(Q_2, D_i)$$

  - **Fuzzy Set:** (generalization to K sub-queries is straight forward)

$$sim(Q_1 \wedge Q_2, D_i) = \min\{sim(Q_1, D_i), sim(Q_2, D_i)\}$$
$$sim(Q_1 \vee Q_2, D_i) = \max\{sim(Q_1, D_i), sim(Q_2, D_i)\}$$

  - **Soft Boolean Operator:** (generalization to K sub-queries is straight forward)

$$sim(Q_1 \wedge Q_2, D_i) = (1-\alpha) \cdot \min\{sim(Q_1, D_i), sim(Q_2, D_i)\} + \alpha \cdot \max\{sim(Q_1, D_i), sim(Q_2, D_i)\} \quad 0 \le \alpha \le 0.5$$
$$sim(Q_1 \vee Q_2, D_i) = (1-\beta) \cdot \min\{sim(Q_1, D_i), sim(Q_2, D_i)\} + \beta \cdot \max\{sim(Q_1, D_i), sim(Q_2, D_i)\} \quad 0.5 \le \beta \le 1$$

  - **Paice-Model:** order the sub-queries in increasing order of their similarity values for AND operator, and order the sub-queries in decreasing order of their similarity values for OR. r is a constant coefficient:

$$sim\left(\bigwedge_{k=1}^{K} Q_k, D_i\right) = \frac{\sum_{k=1}^{K} r^{k-1} \cdot sim(Q_k, D_i)}{\sum_{k=1}^{K} r^{k-1}} \quad \text{with } \forall k, 1 \le k < K\colon sim(Q_k, D_i) \le sim(Q_{k+1}, D_i)$$
$$sim\left(\bigvee_{k=1}^{K} Q_k, D_i\right) = \frac{\sum_{k=1}^{K} r^{k-1} \cdot sim(Q_k, D_i)}{\sum_{k=1}^{K} r^{k-1}} \quad \text{with } \forall k, 1 \le k < K\colon sim(Q_k, D_i) \ge sim(Q_{k+1}, D_i)$$

- **P-Norm-Model:**

$$sim\left(\bigwedge_{k=1}^{K} Q_k\,, D_i\right) = 1 - \sqrt[p]{\frac{\sum_k \left(1 - sim(Q_k, D_i)\right)^p}{K}} \qquad \text{with } 1 \le p < \infty$$

$$sim\left(\bigvee_{k=1}^{K} Q_k\,, D_i\right) = 1 - \sqrt[p]{\frac{\sum_k sim(Q_k, D_i)^p}{K}}$$

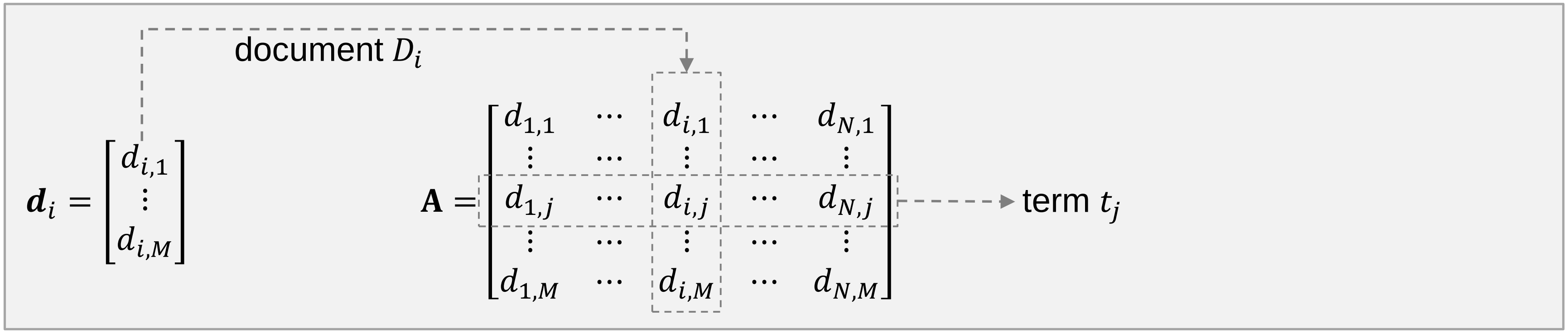
- **Advantages:** simple model with a clean description of query semantics. Very simple to implement and intuitive for users. Even though the definition of query evaluation is rather heuristic, performance is quite good. With the inverted lists method, there is a very efficient way to compute the similarity values. In comparison with the standard Boolean model, we now obtain ranked lists and partial matches, i.e., we can control the size of results to be presented back to the user. Terms are treated differently based on their term occurrence and their discrimination power.
- **Disadvantages:** heuristic similarity scores with little intuition why they work well (no theoretic background for the model). Although the query language is simple, users may find it hard to express a complex information need as a combination of ANDs and ORs. Retrieval quality is ok but other methods (with equal computational complexity) achieve better results.

2.3.3 Vector Space Retrieval

- The vector space retrieval model is by far the most popular of the classic text retrieval models. It has a clean and simple query structure and offers a very fast computational scheme through inverted lists. In contrast to the Boolean models considered so far, it uses the bag-of-words model both to describe the documents and the queries. In other words, a query is considered as a (mini) document and then used as a reference to find similar documents.
- A document D_i is represented as a vector $\boldsymbol{d}_i$ using weighted term frequencies (we do not normalize the term frequencies as with the extended Boolean models):

$$d_{i,j} = tf\left(D_i, t_j\right) \cdot idf\left(t_j\right) \qquad \forall j: 1 \leq j \leq M$$

- All the vectors d_i of the collection $\mathbb{D}$ form the so-called term-document-matrix $\mathbf{A}$ with d_i denoting the i-th column of the matrix, i.e., $a_{j,i} = d_{i,j}$ (the switch of indexes is necessary as d_i is a column vector). A visual representation is as follows:



$$\boldsymbol{d}_i = \begin{bmatrix} d_{i,1} \\ \vdots \\ d_{i,M} \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} d_{1,1} & \cdots & d_{i,1} & \cdots & d_{N,1} \\ \vdots & \cdots & \vdots & \cdots & \vdots \\ d_{1,j} & \cdots & d_{i,j} & \cdots & d_{N,j} \\ \vdots & \cdots & \vdots & \cdots & \vdots \\ d_{1,M} & \cdots & d_{i,M} & \cdots & d_{N,M} \end{bmatrix}$$

It follows that the j-th row in $\mathbf{A}$ contains the information about the term t_j.

- Queries are represented as (very sparse) documents. In other words, the user is not required to enter a complex Boolean query but rather provides a few keywords to search for. A query Q is hence represented as a vector $\boldsymbol{q}$ just like all the documents:

$$q_j = tf\left(Q, t_j\right) \cdot idf\left(t_j\right) \qquad \forall j: 1 \leq j \leq M$$

- We can compute similarity values between documents and queries as a function over the M-dimensional vectors. Two popular methods exists:
  - The **inner vector product** uses the dot-product over vectors to calculate similarity values.

$$sim(Q, D_i) = \boldsymbol{q} \cdot \boldsymbol{d}_i = \sum_{j=1}^{M} q_j \cdot d_{i,j}$$

  We can also represent all similarity values between documents D_i and the query Q as a matrix multiplication:

$$\boldsymbol{sim}(Q, \mathbb{D}) = \begin{bmatrix} sim(Q, D_1) \\ \vdots \\ sim(Q, D_N) \end{bmatrix} = \mathbf{A}^\top \boldsymbol{q}$$

  Note that we only write the above formula for the sake of concise presentation, but we never actually perform matrix multiplications to search for documents. Intuitively, documents are similar to the query if they use the same term as the query (all terms not used in the query have a 0 in $\boldsymbol{q}$). If the query terms are frequently used, high similarity values result. Further we observe that not all query terms are necessary to obtain non-zero similarities (→ partial matches)

- The second measure calculates the **cosine** of the angle between the query vector and the document vector to calculate similarity values.

$$sim(Q,D_i) = \frac{\boldsymbol{q} \cdot \boldsymbol{d}_i}{\|\boldsymbol{q}\| \cdot \|\boldsymbol{d}_i\|} = \frac{\sum_{j=1}^{M} q_j \cdot d_{i,j}}{\sqrt{\sum_{j=1}^{M} q_j^2} \cdot \sqrt{\sum_{j=1}^{M} d_{i,j}^2}}$$

Again, a matrix multiplication leads to all similarity values:

$$\boldsymbol{sim}(Q,\mathbb{D}) = \begin{bmatrix} sim(Q,D_1) \\ \vdots \\ sim(Q,D_N) \end{bmatrix} = \mathbf{L}\mathbf{A}^\top \boldsymbol{q}' \quad \text{with} \quad \mathbf{L} \in \mathbb{R}^{N \times N} = \begin{bmatrix} \frac{1}{\|\boldsymbol{d}_1\|} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\|\boldsymbol{d}_N\|} \end{bmatrix} \quad \text{and} \quad \boldsymbol{q}' = \frac{\boldsymbol{q}}{\|\boldsymbol{q}\|}$$

As before, we only write the above formula for the sake of concise presentation, but we never actually perform matrix multiplications to search for documents. Intuitively, documents are similar to the query if their vectors point to the same direction as the query vector. The number of terms and the weights only play a role to define the direction but the length of the vectors is irrelevant. This provides an equal chance for small and large documents to obtain a high similarity value.

- **Example:** we consider a very simple collection of three documents to observe how the method works. The documents and the query are as follows:

D_1	Shipment of gold damaged in a fire
D_2	Delivery of silver arrived in a silver truck
D_3	Shipment of gold arrived in a truck

Q	gold silver truck

  – We can extract terms and determine document frequencies and inverse document frequencies. The document and query are then represented as vectors ($N = 3, M = 11$):

j	Term t_j	$df(t_j)$	$idf(t_j)$	d_1	d_2	d_3	q
1	a	3	0				
2	arrived	2	.176		.176	.176	
3	damaged	1	.477	.477			
4	delivery	1	.477		.477		
5	fire	1	.477	.477			
6	gold	2	.176	.176		.176	.176
7	in	3	0				
8	of	3	0				
9	silver	1	.477		.954		.477
10	shipment	2	.176	.176		.176	
11	truck	2	.176		.176	.176	.176

To simplify, we use: $idf(t_j) = \log(N) - \log\left(df(t_j)\right)$

$$\underbrace{d_1 \; d_2 \; d_3}_{\mathbf{A}}$$

with inner vector product

$$\mathbf{sim}(\mathbf{Q}, \mathbb{D}) = \begin{bmatrix} .031 \\ .486 \\ .062 \end{bmatrix}$$

$$\Downarrow$$

$$D_2 > D_3 > D_1$$

  - **Observations:** the term-document matrix is usually very sparse, that is a single document only contains a small subset of all possible terms. We also note that we only need to consider the query terms for evaluation; all other terms are eliminated due to the 0-value in $\boldsymbol{q}$. On the other side, a document does not have to contain all query terms to be relevant. In the example before, none of the documents contained all terms. To express such a partial match query with Boolean operators would quickly lead to quite complicated expressions. In the example before, the partial match query in Boolean terms is: (gold AND silver AND truck) OR (gold AND silver) OR (gold AND truck) OR (silver AND truck) OR gold OR silver OR truck.

- **Advantages:** extreme simple an intuitive query model. Very simple to implement and very fast to calculate. Performance is better than with Boolean models and can compete with the best retrieval methods. The model naturally includes partial match queries and documents do not have to contain all query terms to obtain high similarity values.
- **Disadvantages:** heuristic similarity scores with little intuition why they work well (no theoretic background for the model). The similarity measures are not robust and can be biased by authors (spamming of terms). Main assumption of retrieval model is independence of terms which may not hold true in typical scenarios (see synonyms and homonyms). There are several extensions that address this latter aspect.

2.3.4 Probabilistic Retrieval

- The biggest criticism of the models so far is the heuristic approach they take. The methods work and perform well, but there is no foundation to prove correctness. Probabilistic retrieval is a formal approach based on the probability $P(R|D_i)$ that a document D_i is relevant for a query Q and the probability $P(NR|D_i) = 1 - P(R|D_i)$ that a document D_i is not relevant for a query Q. The similarity value is defined as follows:

$$sim(Q, D_i) = \frac{P(R|D_i)}{P(NR|D_i)} = \frac{P(R|D_i)}{1 - P(R|D_i)}$$

- The **Binary Independence Model (BIR)** is a simple technique based on a few assumptions to compute the conditional probabilities above which are

  1. Documents and queries use the set-of-words model (binary vectors)
  2. Terms are independent of each other (the previous models made the same assumptions)
  3. If a term does not appear in the query, it is equally distributed in the relevant and the non-relevant documents (it shall not impact the ranking)

  With these assumptions, we can derive the similarity values. As a first step, we use Bayes' theorem on the definition of the similarity values:

$$sim(Q, D_i) = \frac{P(R|D_i)}{P(NR|D_i)} = \frac{P(D_i|R) \cdot P(R)}{P(D_i|NR) \cdot P(NR)}$$

  We can interpret these new probabilities as follows: $P(R)$ and $P(NR)$ are the probabilities that a randomly selected document is relevant and not relevant, respectively. $P(D_i|R)$ and $P(D_i|NR)$ are the probabilities that D_i is among the relevant and not relevant documents, respectively.

- We now use the assumption that documents are binary vectors and that terms are independent of each other:

Assumption 1: Documents are binary vectors

Assumption 2: Terms are independent

Assumption 1: Documents are binary vectors

$$P(D_i|R) = P(d_i|R) = \prod_{j=1}^{M} P\left(d_{i,j}\middle|R\right) = \prod_{\forall j:\, d_{i,j}=1} P\left(d_{i,j}=1\middle|R\right) \cdot \prod_{\forall j:\, d_{i,j}=0} P\left(d_{i,j}=0\middle|R\right)$$

$$P(D_i|NR) = P(d_i|NR) = \prod_{j=1}^{M} P\left(d_{i,j}\middle|NR\right) = \prod_{\forall j:\, d_{i,j}=1} P\left(d_{i,j}=1\middle|NR\right) \cdot \prod_{\forall j:\, d_{i,j}=0} P\left(d_{i,j}=0\middle|NR\right)$$

- We introduce a short notation for the conditional probabilities on the right most side of the formula above. Let $r_j = P\left(d_{i,j}=1\middle|R\right)$ denote the probability that a relevant document has the term t_j (i.e., $d_{i,j}=1$). Further, let $n_j = P\left(d_{i,j}=1\middle|NR\right)$ denote the probability that a not relevant document has the term t_j (i.e., $d_{i,j}=1$). With that we can write the similarity value as:

$$sim(Q,D_i) = \frac{P(R)}{P(NR)} \cdot \prod_{\forall j:\, d_{i,j}=1} \frac{r_j}{n_j} \cdot \prod_{\forall j:\, d_{i,j}=0} \frac{1-r_j}{1-n_j} \qquad \rightarrow\ sim(Q,D_i) \sim \prod_{\forall j:\, d_{i,j}=1} \frac{r_j}{n_j} \cdot \prod_{\forall j:\, d_{i,j}=0} \frac{1-r_j}{1-n_j}$$

Note that we do not need to compute $P(R)$ and $P(NR)$ as they are depending only on the query and do not change the order of similarity values of documents D_i. Hence, the right formula above is a further simplification that yields the same ranking for documents as the left formula.

- We finally use the third assumption that $r_j = n_j$ if the term t_j does not occur in the query (the term occurs equally likely in the set of relevant and not relevant documents). This means that for all $q_j = 0$, the ratios $\frac{r_j}{n_j}$ and $\frac{1-r_j}{1-n_j}$ are 1 and we can eliminate them from the calculations:

Assumption 3: non-query terms do not impact result

$$sim(Q,D_i) \sim \prod_{\forall j:\, d_{i,j}=1} \frac{r_j}{n_j} \cdot \prod_{\forall j:\, d_{i,j}=0} \frac{1-r_j}{1-n_j} = \prod_{\forall j:\, d_{i,j}=1, q_j=1} \frac{r_j}{n_j} \cdot \prod_{\forall j:\, d_{i,j}=0, q_j=1} \frac{1-r_j}{1-n_j}$$

We drop the condition $d_{i,j} = 1$ in the second product and must compensate in the first product:

$$sim(Q,D_i) \sim \prod_{\forall j:\, d_{i,j}=1, q_j=1} \frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)} \cdot \prod_{\forall j:\ q_j=1} \frac{1-r_j}{1-n_j}$$

Next, we eliminate terms that only depend on the query and do not change the ordering:

$$sim(Q,D_i) \sim \prod_{\forall j:\, d_{i,j}=1, q_j=1} \frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)}$$

We finally obtain a very simple similarity function as a sum over c_j-values. Note that we only need to compute c_j for query terms, that is for a very small number of terms.

$$sim(Q,D_i) \sim \sum_{\forall j:\, d_{i,j}=1, q_j=1} c_j \qquad \text{with } c_j = \log \frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)}$$

- Computing the c_j values: recall that $r_j = P\left(d_{i,j} = 1 \middle| R\right)$ denotes the probability that a relevant document contains the term t_j. Similarly, $n_j = P\left(d_{i,j} = 1 \middle| NR\right)$ denotes the probability that a not relevant document contains the term t_j. To obtain estimates for these probabilities, we ask the user to rate some of the retrieved documents. The more feedback we gather, the better our estimates become. In more details:
  - **Initial step:** without any samples, we assume that query terms are likely to occur in relevant documents while they appear in not relevant documents according to their document frequency. We use the following estimates for the initial step to compute the c_j

$$r_j = 0.5, \qquad n_j = \frac{df\left(t_j\right)}{N} \qquad\qquad \forall j: q_j = 1$$

  - **Feedback step:** although the initial values are a heuristic, we only use them to generate a first result set. We then ask the user to rate the K retrieved documents and annotate them with relevant (R) and not relevant (NR). Let L be the number of documents that the user marked as relevant. Further let k_j be the number of retrieved documents that contain the term t_j (that is the document frequency of t_j over the set of retrieved documents), and let l_j be the number of relevant documents that contain the term t_j (that is the document frequency of t_j over the set of relevant documents). With that, we can estimate new values for r_j and n_j as follows:

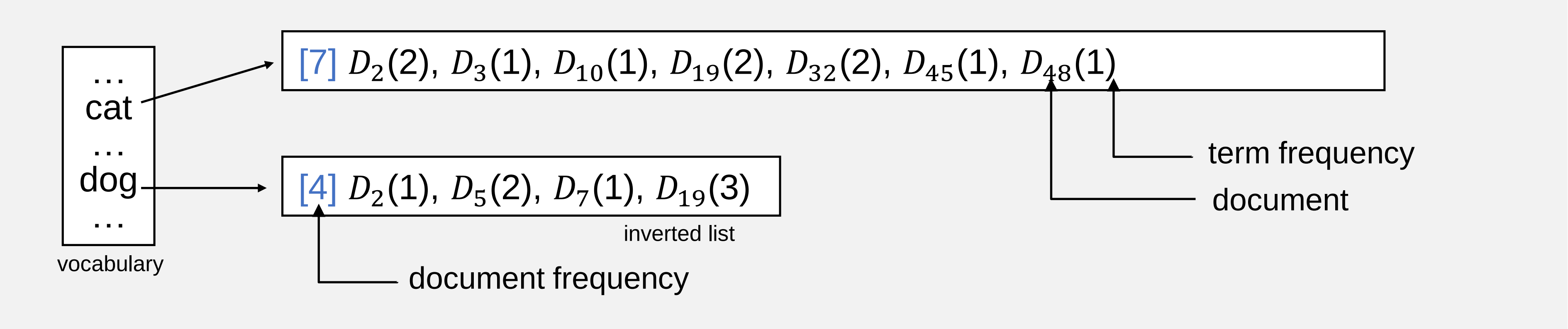
$$r_j = \frac{l_j + 0.5}{L + 1}, \qquad n_j = \frac{k_j - l_j + 0.5}{K - L + 1} \qquad\qquad \forall j: q_j = 1$$

    We use the values 0.5 and 1 in the formula above to prevent numerical issues (0-divisions).

- **Advantages:** the BIR model provides a probabilistic foundation based on simple assumption to define similarity values. The ranking of documents is based on their probability of being relevant for the query. Again, we only require query terms for the calculations of similarity values and, with the inverted lists, we have a very efficient evaluation method at hand. Provides very good performance, especially after a few feedback steps. Also support partial match queries, i.e., not all query terms must occur in relevant documents.
- **Disadvantages:** the simple assumptions do not always hold true. Like discussed in the vector space model, term independence generally does not apply. There are more sophisticated probabilistic models that deal with term dependence, but often come at additional computational overhead. Finally, we note that the ranking of documents does neither take term frequencies nor the discrimination power of terms into account.

2.4 Indexing Structures

- With all retrieval models considered so far, we have observed that ranking (or selection of an answer in Boolean models) only depends on query terms. In addition, if the terms have high discrimination value they are likely to appear only in a few documents. In this section, we look at inverted lists as a simple retrieval model, and apply it to SQL databases for a fast and efficient implementation of text retrieval.
- The term-document matrix is very sparse. We expect that documents only use a small subset of the existing vocabulary, and many terms in the vocabulary occur only in very few documents. Instead of storing the full matrix, we keep condensed rows for each term. For example, we have two terms "dog" and "cat" which appear in some document. In addition, we want to keep track of term frequencies in the documents to apply a more sophisticated ranking function. A typical inverted list looks then something like this:


…
cat
…
dog
…
vocabulary
[7] D_2(2), D_3(1), D_{10}(1), D_{19}(2), D_{32}(2), D_{45}(1), D_{48}(1)
[4] D_2(1), D_5(2), D_7(1), D_{19}(3)
inverted list
document frequency
term frequency
document

- **Application to standard Boolean model:** we can calculate the result with set operations over the atomic parts of the query (must contain term, or must not contain term). The inverted lists provide the sets for the atomic parts "must contain terms", and, with some restrictions, we can also use them for "must not contain terms". For example:
  - $Q =$ "cat" AND "dog"
    - $\mathbb{S}_{cat} = \{D_2, D_3, D_{10}, D_{19}, D_{32}, D_{45}, D_{48}\}$, $\mathbb{S}_{dog} = \{D_2, D_5, D_7, D_{19}\}$
    - $\mathbb{Q} = \mathbb{S}_{cat} \cap \mathbb{S}_{dog} = \{D_2, D_{19}\}$
  - $Q =$ "cat" AND (NOT "dog")
    - $\mathbb{S}_{cat} = \{D_2, D_3, D_{10}, D_{19}, D_{32}, D_{45}, D_{48}\}$, $\mathbb{S}_{dog} = \{D_2, D_5, D_7, D_{19}\}$
    - $\mathbb{Q} = \mathbb{S}_{cat} - \mathbb{S}_{dog} = \{D_3, D_{10}, D_{32}, D_{45}, D_{48}\}$
  - More generally, NOT-clauses are only allowed within AND-clauses (translates into minus set-operation), but not in OR-clauses. A query like: "cat" OR (NOT "dog) cannot be answered with only the inverted lists; in addition, such a query is not really meaningful. So the restriction is hardly relevant for users.
  - To accelerate the set operations, we sort the inverted lists by increasing document frequencies. This way the intermediate results sets are smaller.
- **Retrieval Models with ranking:** all the models with ranking that we considered so far, have a partial match capability. In other words, we must retrieve all documents that contain at least one query term and then evaluate the similarity values only for these selected documents. For example:
  - $Q =$ "cat dog" (vector space retrieval, probabilistic retrieval)

    $Q =$ "cat AND dog" $Q =$ "cat AND (NOT dog)" (extended Boolean model)
    - $\mathbb{S} = \mathbb{S}_{cat} \cup \mathbb{S}_{dog} = \{D_2, D_3, D_5, D_7, D_{10}, D_{19}, D_{32}, D_{45}, D_{48}\}$

- The typical implementation stores the inverted lists as individual files. But we can also efficiently implement inverted lists in a SQL database and exploit other features that a database provides (proven storage, transaction management, high availability, disaster recovery, …). For the implementation, we need the tables: 1) document table, 2) vocabulary, 3) inverted list (here: index). In addition, we require a (temporary) query table to simplify SQL queries.

Document

docid	doc_name	date	dateline
1	WSJ870323-0180	3/23/87	Turin, Italy
2	WSJ870323-0161	3/23/87	Du Pont Company, Wilmington, DE

Index

doc_id	term	tf
1	commercial	1
1	vehicle	1
1	sales	2
1	italy	1
1	february	1
1	year	1
1	according	1
	...	
2	krol	2
2	president	2
2	diversified	1
2	company	1
2	succeeding	1
2	dale	1
2	products	2
	....	

Term

term	idf
according	0.9031
commercial	1.3802
company	0.6021
dale	2.3856
diversified	2.5798
february	1.4472
italy	1.9231
krol	4.2768
president	0.6990
products	0.9542
sales	1.0000
succeeding	2.6107
vehicle	1.8808
year	0.4771

QUERY

term	tf
vehicle	1
sales	1
italy	1

- Evaluation of a Boolean Query
  - Option 1: no Query table
    $Q =$„vehicle sales italy“

```
SELECT a.DocID
  FROM Index a,Index b,Index c
 WHERE a.Term=‘vehicle‘ AND
       b.Term=‘sales‘ AND
       c.Term=‘italy‘ AND
       a.DocID=b.DocID AND
       a.DocID=c.DocID;
```

  - Option 2: with Query table
    $Q =$„vehicle sales italy“

```
DELETE FROM Query;
INSERT INTO Query
      VALUES(‘vehicle‘,1);
INSERT INTO Query
      VALUES(‘sales‘,1);
INSERT INTO Query
      VALUES(‘italy‘,1);


SELECT i.DocID
  FROM Index i, Query q
 WHERE i.Term=q.Term
 GROUP BY i.DocID
HAVING COUNT(i.Term)=
    (SELECT COUNT(*) FROM QUERY)
```

- Evaluation with Vector Space Retrieval
  - Example: inner vector product
    Q =„vehicle sales italy“

```
DELETE FROM Query;
INSERT INTO Query
         VALUES(‘vehicle‘,1);
INSERT INTO Query
         VALUES(‘sales‘,1);
INSERT INTO Query
         VALUES(‘italy‘,1);


SELECT i.DocID, SUM(q.tf * t.idf * i.tf * t.idf)
  FROM Query q, Index i, Term t
 WHERE q.Term=t.Term AND
       i.Term=t.Term
 GROUP BY i.DocID
 ORDER BY 2 DESC;
```

2.5 Lucene - Open Source Text Search

- Apache hosts several projects to provide easy to use yet powerful text and web retrieval. All of them are based on the core engine called *Lucene.* In addition, third-party libraries enrich Lucene with additional content extractor and analyzers.
  - Lucene: core retrieval library for both analysis of documents and searching
  - Apache Tika: parsers and extractors for various file formats
  - Nutch: open source web search engine with scalable, distributed crawlers and a Tomcat web application to search through the content
  - Solr: open source enterprise search engine for a rich set of file formats
  - Elasticsearch: an enterprise search server

- In this chapter, we look at:
  - how Lucene analyzes documents
  - how Lucene ranks documents
  - how to use Lucene in own applications

- Note: this is not meant to be a complete overview of Lucene. Refer to the online documentation or to books such as “Lucene in Action” to get more details

2.5.1 History of Lucene

- Lucene started as a SourceForge project and joined the Apache Jakarta family in 2001. Original author was Doug Cutting. Since 2005, Lucene is a top-level Apache project with many sub-projects. Some of them, namely Nutch and Tika, have become independent Apache projects.

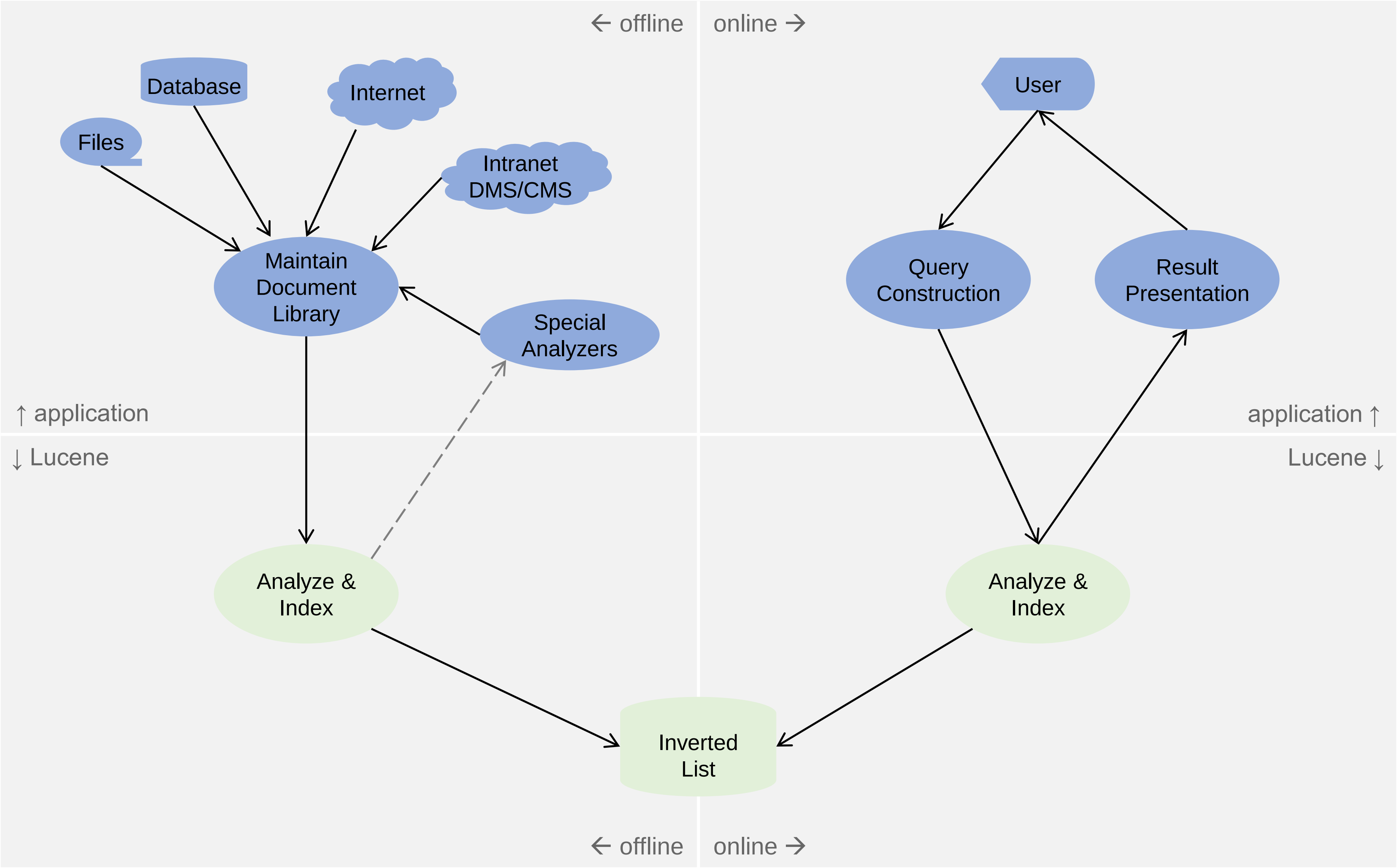
- Main versions introduced (selected versions):
  - 1.01b (July 2001): last SourceForge release
  - 2.0 (May 2006): clean up of code, removed deprecated methods
  - 3.0 (November 2009): cleanup and migration to Java 1.5 (generics, var args)
    3.6 is latest build released on July, 2012
  - 4.0 (August 2012): speedup of indexing and retrieval
  - 5.0 (February 2015): index safety, many adjustments on the API
  - 6.0 (April 2016): Java 8, classification, spatial module update
  - 7.0 (September 2017): Java 9 and support of Jigsaw modularization
  - 7.5 (September 2018): Integration of OpenNLP

- Lucene implementations
  - Java (original), C++ (CLucene), .NET (Lucene.NET), C (Lucene4c), Objective-C (LuceneKit), Python (PyLucene), PHP 5 (Zend), Perl (Plucene), Delphi (MUTIS), JRuby (Ferret), Common Lisp (Montezuma)

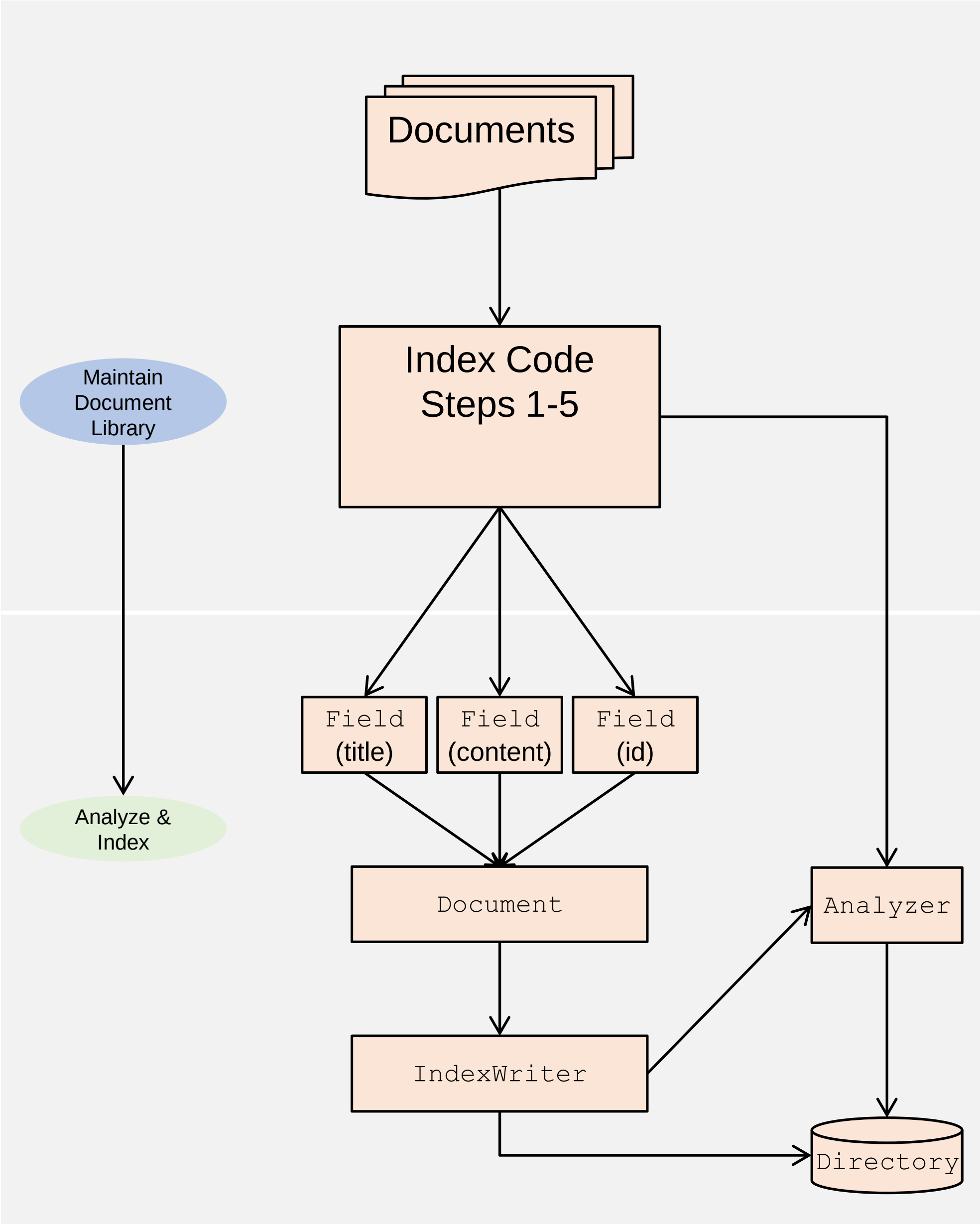
2.5.2 Core Data Model of Lucene

- Lucene is a high-performance, full-featured text search **library**. It is suitable for a wide range of applications that require text retrieval functions. Most importantly, it works across different platforms, firstly due to its Java implementation, and secondly, due to the many ports to other programming languages.
- If you are looking for an open source search engine, Lucene based projects such as Nutch (web search engine) or Solr (enterprise search engine) provide ready-to-deploy search applications. In all other cases, we have to implement the search features through the Lucene APIs.
- The core concepts of Lucene revolve around
  - `Document` and `Field` to encompass the content of documents
  - `Analyzer` to parse the content and extract features
  - `IndexWriter` which maintains the inverted index including concurrency control
  - `Directory` that holds the inverted index structures
  - `Query` and `QueryParser` represent queries and parse input strings, respectively
  - `Term` and `TermQuery` denote unit search expressions
  - `IndexSearcher` exposes search methods over the inverted indexes
  - `TopDocs` contains the result of a search sorted by scores

- Lucene’s API is split into offline analysis functions and online search function. The interaction with an application is as follows:



2.5.3 Indexing Documents with Lucene



1. Select Directory to store Index in

   ```
   directory = FSDirectory.open("./index");
   ```

2. Create Analyzer for Documents

   ```
   analyzer = new StandardAnalyzer();
   ```

3. Create Document and add Fields

   ```
   doc = new Document();
   doc.add(new TextField("title",title,
                         TextField.TYPE_STORED));
   doc.add(new TextField("content",content,,
                         TextField.TYPE_NOT_STORED));
   doc.add(new StoredField("id",id));
   ```

4. Get Index Writer and add Document

   ```
   config = new IndexWriterConfig(anaylzer);
   writer = new IndexWriter(directory,config);
   writer.addDocument(doc);
   ```

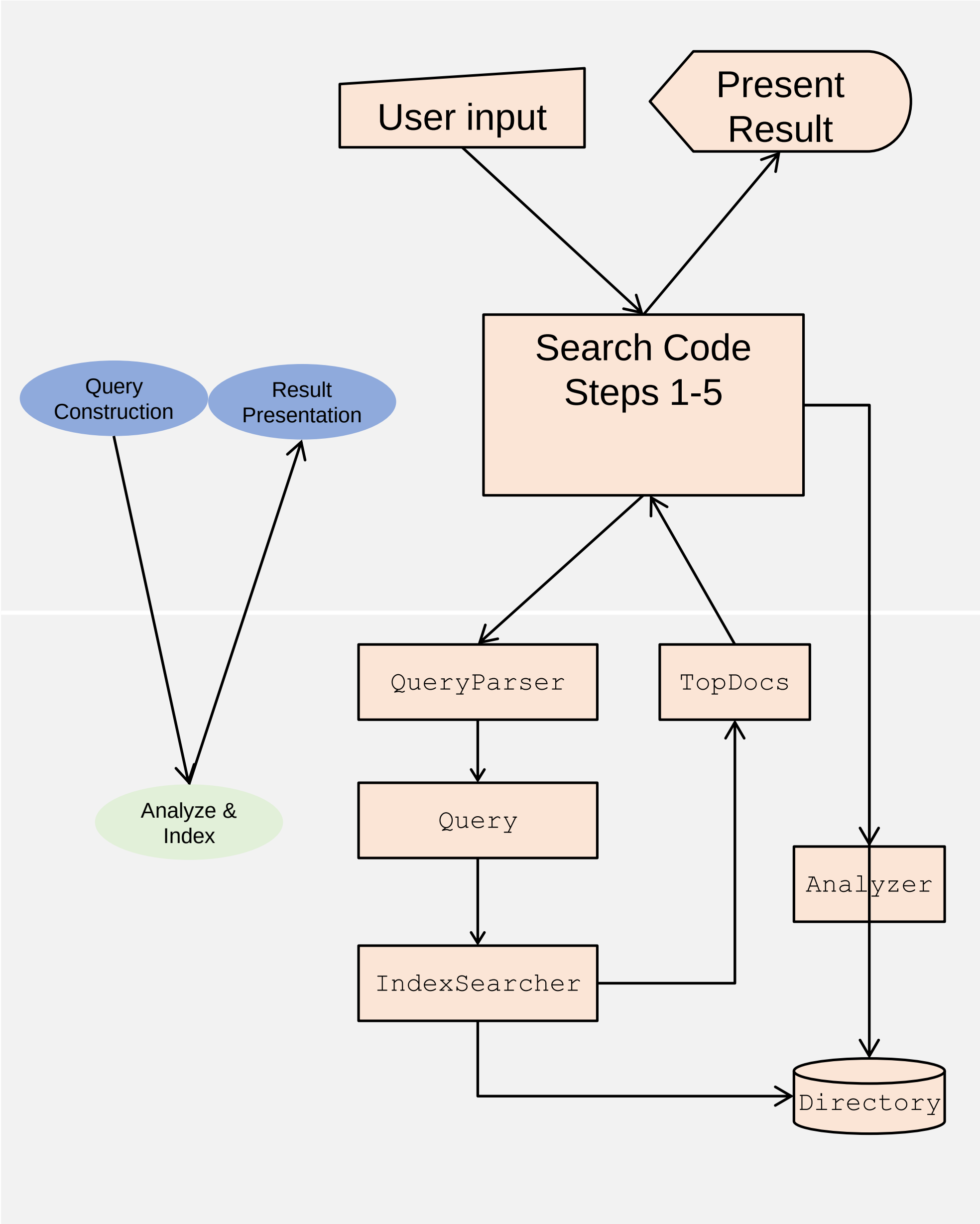
5. Close Index Writer (optionally optimize)

   ```
   writer.optimize();
   writer.close();
   ```

2.5.4 Indexing Documents with Lucene

- Directory
  - Lucene provides multiple ways to maintain and persist inverted indexes. Among them are file based indexes, memory based indexes, and database indexes
  - The `LockFactory` associated with a directory implements basic concurrency control mechanisms. `IndexWriter` and `IndexSearcher` provide concurrency control to the application to ensure integrity of the indexes (other transaction attributes depend on the selected directory implementation)
- Analyzers
  - Lucene and 3rd party extensions provide a rich set of pre-defined analyzers with support for various languages. The main function of an analyzer is to return a `TokenStream`. A token stream implements a pipeline that cascades a tokenizer with a set of token filters.
  - A `Tokenizer` parses the fields of documents, removes syntactical elements, and produces a stream of tokens.
  - A `TokenFilter` filters/changes/aggregates elements in the token stream. Prominent examples include stemming, stop word elimination, and lower case converter.
- Fields
  - Lucene is able to store additional attributes for each document in the index. The purpose of fields is two-fold:
    - Ability to restrict the search on specified meta data items (e.g., only title, author, abstract, etc.)
    - Ability to store data that identify the document (or are relevant for presentational purposes)
  - Creation of fields includes many options (newer release subsumes all of them in `FieldType`)
    - `Field.Store`: `YES` or `NO` indicating whether the content needs to be stored. `NO` means that the content is only analyzed but not available at search time any more. Use `YES` for identifying attributes (or for presentation). Typical examples include ID, file name, document type, date of insertion, size of document.
    - `(deprecated) Field.Index`: main values are `ANALYZED` and `NO`. `NO` indicates that the field must not be analyzed; it is not possible to search for such attributes. `ANALYZED` is used for content that must be indexed.
    - `(deprecated) Field.TermVector`: allows to fine tune what term vector information is kept in the index.

2.5.5 Searching Documents with Lucene



1. Select Directory where Index resides

```
directory = FSDirectory.open("./index");
```

2. Create Analyzer as used for Documents

```
analyzer = new StandardAnalyzer();
```

3. Create Query (optionally through QueryParser)

```
parser = new QueryParser("content", analyzer);
Query query = parser.parse(queryStringFromUser);
```

4. Get Index Searcher and Search

```
searcher = new IndexSearcher(directory);
TopDocs hits = searcher.search(query, NUM_RESULTS);
```

5. Present Result

```
for(int i=0;i<hits.scoreDocs.length;i++){
  doc = searcher.doc(hits.scoreDocs[i].doc);
  System.out.printf("  %4d %1.3f  %s %s\n",
       i+1,
       hits.scoreDocs[i].score/hits.getMaxScore(),
       doc.get("id"), doc.get("title"));
}
```

2.5.6 Searching Documents with Lucene

- Query and QueryParser
  - Lucene provides multiple ways to query the content of an index. Queries are always against the content of analyzed field data. Atomic queries consist of term queries, range queries, phrase queries, fuzzy queries (searching for all terms that are close to the given one), wildcard queries, and so on. Atomic queries can be combined by means of Boolean operators.
  - QueryParser simplifies the interface with a standard way how users have to enter queries
    - Query is a set of clauses optionally prefixed with '+' (must include) and '-' (must not be included)
    - A clause can be a single term such as 'hello' for the default search field, a search term for a selected field such as 'title:hello', a fuzzy query such as 'hello~' or-ing all similar terms in the index, a wildcard-query such as 'h?llo' or-ing all matching terms, and many more.
  - Scores are computed through a Similarity object. The example code uses the default scoring, but it is possible to overwrite how Lucene scores and ranks documents (see next slide)
- TopDocs
  - The search method of the IndexSearcher returns the top (NUM_RESULTS) documents matching the query and ordered by their score.
  - Retrieval of the content of fields of document is through the IndexSearcher. TopDocs only holds Lucene internal document identifiers (property doc of scoreDocs field in TopDocs).
  - Only fields that were indexed with Field.Store.YES can be retrieved after a search. Any other metadata has to be retrieved by the application it self.
- Analyzer
  - Use the same analyzer object as for indexing the documents in offline mode. Lucene provides versioned standard analyzer to avoid confusion should the standard implementation change over time.

2.5.7 Retrieval Model of Lucene

- Lucene combines Boolean retrieval with vector space retrieval. Only documents that match the Boolean query are returned. The candidates are scored with an extended version of the vector space retrieval model, and the top k documents are returned. The following discusses the standard similarity scoring scheme, but you can change and tune many aspects of it (see JavaDoc for class `org.apache.lucene.search.Similarity`).
- Boolean Retrieval Part
  - Applications can define arbitrary Boolean expressions on fields content with
    - atomic queries such `TermQuery`, `RangeQuery`, or any other `Query`
    - and a Boolean clause constraint whether `MUST`, `MUST NOT`, or `SHOULD` occur
  - Example: +information –multimedia retrieval search

```
TermQuery q1 = new TermQuery(new Term("content","information"));
TermQuery q2 = new TermQuery(new Term("content","retrieval"));
TermQuery q3 = new TermQuery(new Term("content","search"));
TermQuery q4 = new TermQuery(new Term("content","multimedia"));
BooleanQuery query = new BooleanQuery();
query.add(q1, BooleanClause.Occur.MUST);
query.add(q2, BooleanClause.Occur.SHOULD);
query.add(q3, BooleanClause.Occur.SHOULD);
query.add(q4, BooleanClause.Occur.MUST_NOT);
```

  - `FuzzyQuery` and `WildcardQuery` translate into a `MultiTermQuery` over a set of terms
    - `FuzzyQuery` ('hello~0.5') expands to a search over all terms in the index that have a normalized similarity of 0.5 and larger (value btw 0 and 1). Similarity is measured with edit distance and normalized over the length of the term.
    - `WildcardQuery` ('h?llo') expands to a search over all terms that match the pattern

- Boolean Retrieval Part (contd)
  - `IndexSearcher` uses the inverted lists in the directory to retrieve all documents that match the Boolean condition. This is the set of candidates.

- Ranking uses an extended version of the cosine measure. However, there are several additional factors and normalizations built into the standard similarity measure
  - The conceptual scoring formula is:

$$\mathrm{score}(q,d) = \mathrm{coord\_factor}(q,d) \cdot \mathrm{query\_boost}(q) \cdot \frac{V(q)\cdot V(d)}{\|V(q)\|} \cdot doc\_len\_norm(d) \cdot doc\_boost(d)$$

    - $\mathrm{coord\_factor}(q,d)$: score factor based on how many query terms are found in the document. In essence, this scores how many of the optional terms (OR clauses) are found in d.
    - $\mathrm{query\_boost}(q)$: boost factor for individual query terms to be taken into account
    - $V(q), V(d)$: vector representation, i.e., tf*idf weighted number of term occurrences
    - $doc\_len\_norm(d)$: unlike the normalization of queries by their length, documents are normalized by the length of a field (number of term occurrences) to boost smaller fields over larger fields
    - $doc\_boost(d)$: application specified boost factor defined at document insertion time

  - To simplify computation, Lucene's implementation is as follows
    - query norm and query boost are combined as they are known at search start time
    - document norm and document (filed) boost values are stored in the index for each term

- Ranking in Lucene (contd)
  - The formula defined by `TFIDFSimilarity` is:

$$score(q,d) = coord(d,q) \cdot queryNorm(q) \cdot \sum_{t\ in\ q} \left(tf(t,d) \cdot idf(t)^2 \cdot boost(t) \cdot norm(t,d)\right)$$

- $coord(d,q) = \frac{overlap}{\max\_overlap}$ boosts documents that contain more of the query terms (not the number of occurrences. max_overlap is the maximum number of query terms found in a single document.

- $queryNorm(q) = \frac{1}{\left(boost(q)^2 \cdot \sum_{t\ in\ q}(idf(t) \cdot boost(t))^2\right)}$ is used to make scores of different (sub-)queries comparable. It does not affect document ranking (constant factor) but how a query overall is weighted. $boost(q)$ is an application specified boost factor for the query.

- $tf(t,d) = \sqrt{frequency\ of\ t\ in\ d}$ documents with higher numbers of term occurrences obtain a higher weight. Note the query term occurrences are not taken into account. Rather, Lucene treats each term occurrence the same, e.g., if the term occurs twice, two sub-queries exist for weighting

- $idf(t) = 1 + log\left(\frac{numDocs}{docFrequency+1}\right)$ denotes the standard inverse document frequency applied to both query and document terms

- $boost(t)$ represents an application specified boost value for a term t in the query

- $norm(t,d) = boost(d) \cdot \frac{1}{number\ of\ terms\ in\ field} \cdot \prod_{field\ f\ in\ d\ anmeds\ as\ t} boost(f)$ denotes a value that Lucene computes at indexing time and stores within the inverted lists for each term in document d. $boost(d)$ denotes a boost factor that applications can specify when adding documents.

2.6 Literature and Links

General Books on Text Retrieval

- Gerard Salton and Michael J. McGill. Information Retrieval - Grundlegendes für Informationswissenschaftler, McGraw-Hill Book Company, 1983.
- W.B. Frakes and R. Baeza-Yates. Information Retrieval, Data Structures and Algorithms, Prentice Hall, 1992.
- Karen Sparck Jones and Peter Willet. Readings in Information Retrieval. Morgan Kaufmann Publishers Inc., 1997.
- David A. Grossmann and Ophir Frieder. Information Retrieval: Algorithms and Heuristics, Kluwer Academic Publishers, 1998 (1st edition), 2004 (2nd edition).
- Ricardo Baeza-Yates and Berthier Ribeiro-Neto. Modern Information Retrieval, ACM Press Books, 1999 (1st edition), 2011 (2nd edition).
- Sandor Dominich. Mathematical Foundations of Information Retrieval, Kluwer Academic Publishers, 2001.
- Christopher Manning, Prabhakar Raghavan, Hinrich Schütze. Introduction to Information Retrieval, Cambridge University Press, 2008
- Stefan Büttcher, Charles Clarke, Gordon Cormack. Information Retrieval - Implementing and Evaluating Search Engines. MIT Press 2010.
- Steven Bird, Ewan Klein, and Edward Loper. Natural Language Processing with Python. O'Reilly Media, 2009. Free online version: http://www.nltk.org/book/

Thesaurus & Ontologies for selected Languages

- EuroWordNet: http://www.illc.uva.nl/EuroWordNet/
- GermanNet: http://www.sfs.uni-tuebingen.de/lsd/
- WordNet: http://www.cogsci.princeton.edu/~wn/

Implementations

- Natural Language Toolkit (NLTK), http://www.nltk.org
- Apache Lucene, https://lucene.apache.org