Information Retrieval

Prof. Dr. Axel Ngonga

Student lecture notes by
Tanja Tornede & Alexander Hetzer
Status: July 27, 2018

Contents

Chapter 1

Boolean Retrieval

1.1 Definition: Information Retrieval

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

1.2 Idea: Boolean Model

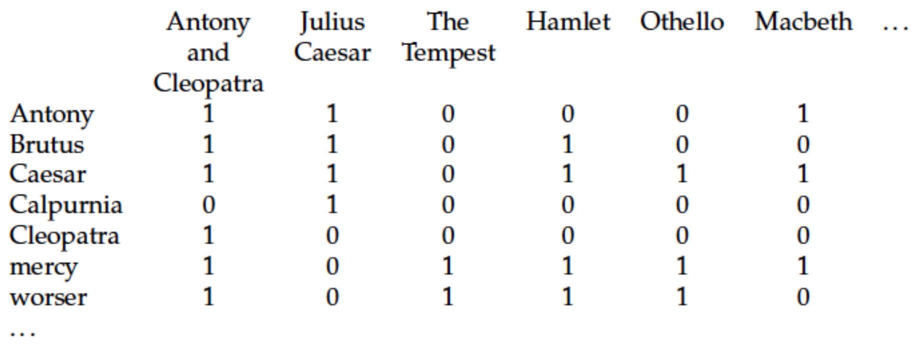
- Queries are boolean expression, e.g. `Caesar AND Brutus`
- Search engine returns all documents satisfying the boolean expression (`AND, OR, NOT`)
  - Views each document as a **set** of terms
  - Is precise: document matches condition or not (no fuzziness)
- Many professional searchers (e.g. lawyers) like Boolean queries, as you know exactly what you get

1.3 Approach: Linear Scan (Grepping)

- Idea: run through all documents and include all the ones having the favorable characteristics and exclude all one have non-favorable characteristics
- Problems:
  - Slow (for large collections)
  - Grep is line-oriented whereas IR is document-oriented
  - `NOT Calpurnia` is a non-trivial task
  - Other operations (e.g. `Romans NEAR countryman`) not feasible
  - Does not allow ranked retrieval
- Solution: **Index** documents before searching

1.4 Binary Term-Document Incidence Matrix

- For each word in the vocabulary of your corpus and for each document store whether the word is contained in the document or not
- Words can be generalized by **terms** (normalized words or compound words)
  - Column: vector of a document giving information about which terms it contains
  - Row: vector of a term giving information about in which documents it is contained



	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

▶ **Figure 1.1** A term-document incidence matrix. Matrix element (t, d) is 1 if the play in column d contains the word in row t, and is 0 otherwise.

- Problem:
  - Consider $n = 10^6$ documents with 1000 tokens and $M = 500000$ distinct terms in the collection

  ⇒ 10^9 tokens in total, with 6 bytes per token ⇒ 6 GB of space
  - Incidence matrix would have the dimensions 500000×10^6
    - Matrix is extremely sparse though, i.e. at most 1 billion 1 entries (as each document has only 1000 tokens)

    ⇒ Storing only the 1's is enough ⇒ **Inverted Index**
  - Note that we are making a term/token distinction

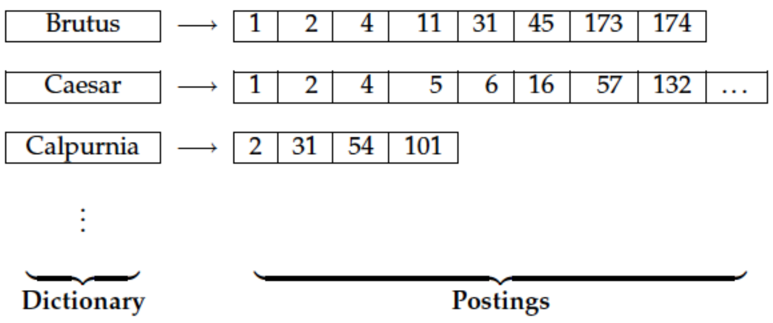
1.5 Notation

- **Documents**: units the retrieval system is working on (e.g. wikipedia articles, websites)
- **Document collection/Corpus**: group of documents over which we perform retrieval
- **Token**: one "word" in a document
- **Term**: one "word" in the vocabulary (i.e. normalized token)
- **Ad hoc retrieval task**: system aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query.
- **Information need**: the topic about which the user desires to know more

- **Query**: what the user conveys to the computer in an attempt to communicate the information need
- Document is **relevant**:it is one that the user perceives as containing information of value with respect to their personal information need
- **Effectiveness** of an IR system: the quality of its search results. Effectiveness statistics:
  - **Precision**: fraction of the returned results which are relevant (to the information need)
  - **Recall**: fraction of the relevant documents in the collection which were returned by the system

1.6 Inverted Index

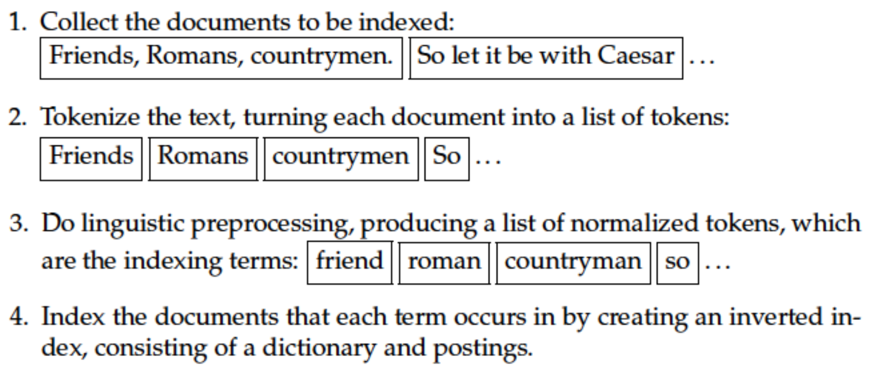
- keep a **dictionary** (vocabulary) of terms, usually sorted alphabetically


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

▶ **Figure 1.3** The two parts of an inverted index. The dictionary is commonly kept in memory, with pointers to each postings list, which is stored on disk.

- for each term t store a list of documents containing term t
  - list should be sorted by document id (for efficient operations)
  - list is called **postings list**
  - single entry is called **posting**
  - all postings lists together are called **postings**

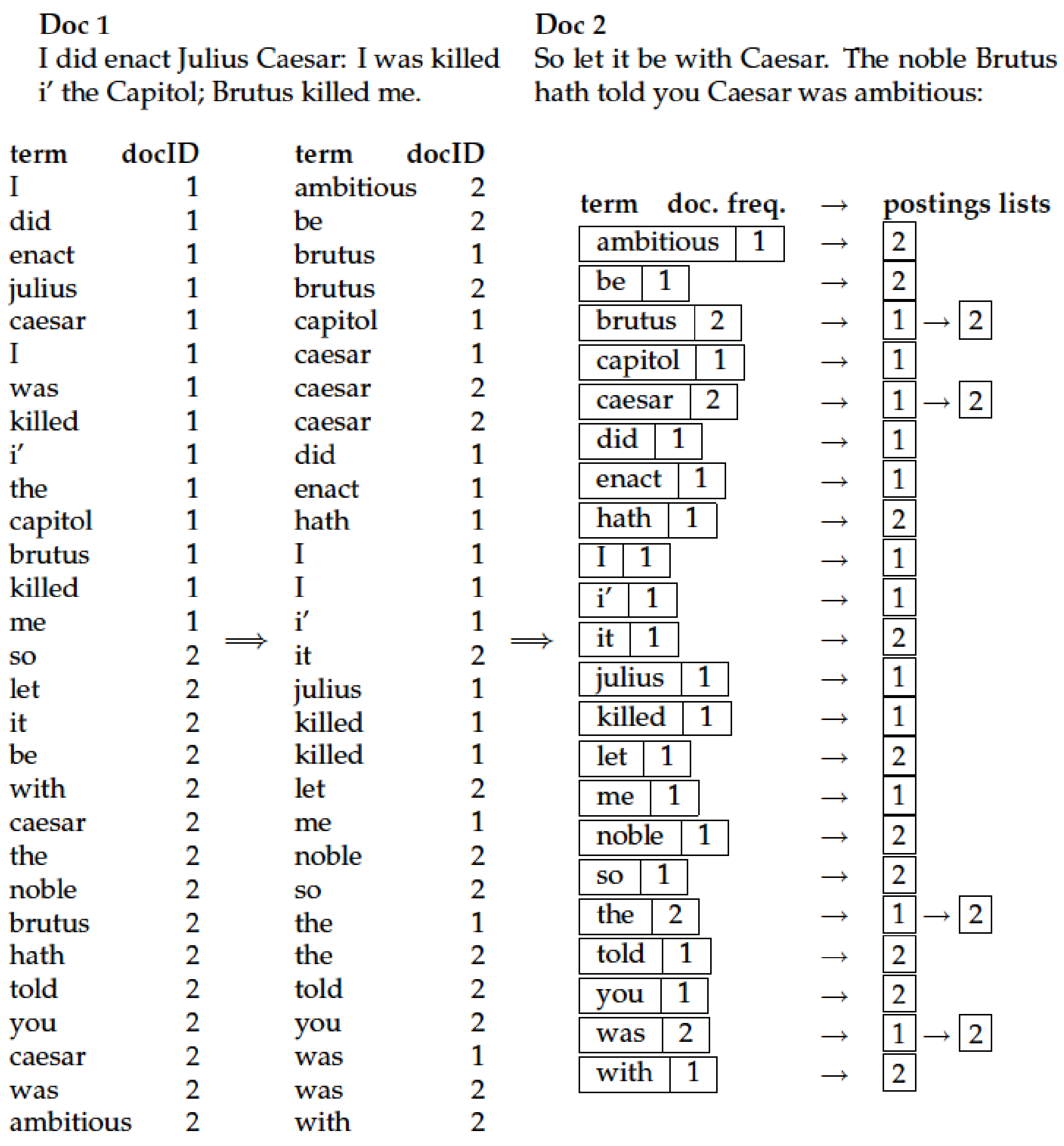
1.6.1 Index Construction



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

Step 4 in Detail

1) Generate postings
2) Sort postings (alphabetically)
3) Create postings list (sorted by document id) and determine document frequency (number of documents the term occurs in)
4) Split result into dictionary and postings list



Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

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

$\Longrightarrow$

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

$\Longrightarrow$

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Remarks

- sorting and statistics (document frequency) mainly useful for efficient query resolution
- dictionary is commonly kept in memory (as usually smaller than postings) whereas postings lists are kept on disk

1.7 Query Processing

- Efficient intersection operation is crucial!

Conjunctive Queries (Efficient Intersection)

- Assumes that the postings lists are sorted wrt. the document id

```
INTERSECT($p_1, p_2$)
answer $\leftarrow \langle\ \rangle$
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then ADD($answer, docID(p_1)$)
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
              then $p_1 \leftarrow next(p_1)$
              else $p_2 \leftarrow next(p_2)$
return answer
```

- Runtime: $O(|p_1| + |p_2|)$
- Formal querying complexity (linear scan): $\Theta(N)$, where N is the number of documents in the collection

⇒ No difference in the O-calculus, as we only gain a (in practice huge) constant

- Algorithm can be extended to more complex query for `AND NOT`

Algorithm 2 AND NOT(x,y)

```
result $\leftarrow \emptyset$
$p_x \leftarrow$ postings list of $x$
$p_y \leftarrow$ postings list of $y$
while $p_x \neq NIL$ do
    if $docID(p_x) = docID(p_y)$ then
        $p_x \leftarrow next(p_x)$
        $p_y \leftarrow next(p_y)$
    else if $docID(p_x) < docID(p_y) \vee p_y \neq NIL$ then
        $ADD(result, docID(p_x))$
        $p_x \leftarrow next(p_x)$
    else
        $p_y \leftarrow next(p_y)$
    end if
end while
return result
```

1.8 Query Optimization

- Major element of optimization for boolean queries: order in which the postings lists are accessed
    - Start with the shortest postings list, then keep cutting further

1.8.1 Conjunctive Queries (Optimized)

- consider a query which is an AND of $n > 2$ terms
- **effective strategy**: process in order of increasing document frequency (i.e. number of list entries)
    - start with the smallest postings list and AND it with the next smallest, etc.
    - then all intermediate results will not be bigger than the smallest postings list

```
INTERSECT($\langle t_1, \ldots, t_n \rangle$)
  $terms \leftarrow$ SORTBYINCREASINGFREQUENCY($\langle t_1, \ldots, t_n \rangle$)
  $result \leftarrow postings(first(terms))$
  $terms \leftarrow rest(terms)$
  while $terms \neq$ NIL and $result \neq$ NIL
  do $result \leftarrow$ INTERSECT($result, postings(first(terms))$)
      $terms \leftarrow rest(terms)$
  return $result$
```

▶ **Figure 1.7** Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms.

1.8.2 More General Optimization

- Example: `(madding OR crowd) AND (ignoble OR strife)`
- Strategy:
    - Get frequencies of all terms (i.e. madding, crowd, ignoble and strife)
    - Estimate the size of each OR by the sum of its frequencies (conservatively, i.e. we compute an upper bound on the disjunction)
    - Process in increasing order of OR sizes

CHAPTER 2

Term Vocabulary and Postings Lists

- **tokenization**: process of chopping character streams into tokens
- **linguistic preprocessing**: building equivalence classes of tokens which are the set of tokens to be indexed

2.1 Document Unit, Format and Language Complications

- document unit depends on situation (e.g. email with attachments - how many docs)
- documents can contain several languages
- determining the file format can be problematic
- determining the character set can be problematic too

2.2 Definitions

Word A delimited string of characters as it appears in the text

Term A "normalized" word (case, morphology, spelling, etc.); an equivalence class of words

Token An instance of a word or term occurring in a document

Type The same as a term in most cases: an equivalence class of tokens

2.3 Normalization

- need to "normalize" words in documents as well as query terms into the same form (e.g. U.S.A vs USA)
- most common way: define **equivalence classes** of terms
  - can be done implicitly by removing characters like hyphens
  - BUT: only easy when removing characters (but not when adding chars)
- alternative: **asymmetric expansion**

- examples
  - window → window, windows
  - windows → Windows, windows
  - Windows (no expansion)
- idea: create expansion lists of different terms which can overlap **without being identical**
- more powerful but less efficient than equivalence classes

- normalization and language detection interact (mit Hund vs. go to M.I.T)

2.4 Tokenization

- should use same tokenization on documents and queries
- tokenization is language specific
- depending on application, special tokens can be ignored
- common problems
  - one or two word problem: "San Francisco", "data base", "Hewlett-Packard"
  - numbers
  - segmentation problem / missing whitespace (mostly for languages like Inuit, etc.)
  - multiple alphabets in same language (Japanese)
  - different writing directions (Arabic)
    - text and numbers are written in different directions
  - accents and umlauts
    - usually removed, but can be problematic

2.4.1 Case Folding

- reduce all letters to lower case
- even though case can be semantically meaningful, it is often best to lowercase everything since users will lowercase regardless of correct capitalization

2.4.2 Stop Words

- **stop words**: extremely common words which appear to be of little value in helping documents matching a user need (e.g. "of", "the", "he")
- stop word elimination used to be standard
- BUT: stop words are needed for phrase queries : "King of Denmark"
- most web search engines index stop words

2.5 Stemming and Lemmatization

- documents use different forms of words (e.g.: organize, organizing, organizes)
- overall goal of stemming and lemmatization: reduce inflectional form and sometimes derivationally related forms of a word to a common base form

2.5.1 Lemmatization

- process uses a vocabulary and morphological analysis of words
- aims at removing inflectional endings only in order to return the base form of a word (**lemma**)

2.5.2 Stemming

- crude **heuristic process** that **chops off end of words** (in the hope to be right most of the time)
- language dependent
- most common algorithm: **Porter's stemmer**
- in general: stemmer use language specific rules but require less knowledge than a lemmatizer

Porter's Stemmer

- empirically very effective
- consists of five phases of word reduction, applied sequentially
- in each phase: conventions to select rules

2.5.3 Effectiveness

- in English both forms help only a little
- stemming increases recall while harming precision as a lot of special information is lost (e.g. operations research $\rightarrow$ oper res)
- lemmatization is a lot better in languages with more morphology

2.6 Skip Pointers: Faster Postings List Intersection

- basic intersection algorithm requires linear time $O(m+n)$ (m / n: postings list lengths)
- idea: use **skip lists** to perform intersection in sub-linear time

  $\Rightarrow$ add skip pointers at indexing time
- skip pointers are shortcuts in order to avoid processing unimportant parts of the postings lists
- main questions:
  - where to place skip pointers?
  - how to do merging using skip pointers?

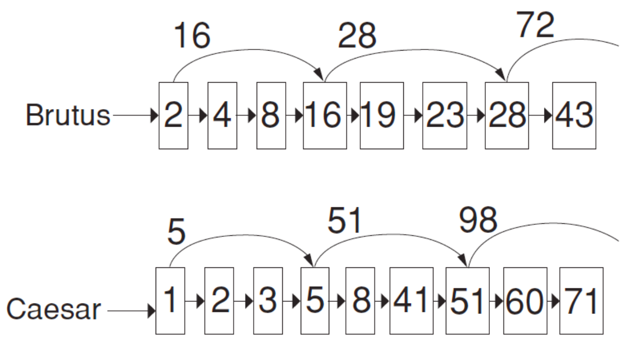
2.6.1 Intersecting with Skip Pointers

- when advancing a pointer, check if taking the skip pointer (if one is present) can be used without going too far (still $\leq$ than other pointer)

```
IntersectWithSkips($p_1, p_2$)
 1  $answer \leftarrow \langle\,\rangle$
 2  while $p_1 \neq$ NIL and $p_2 \neq$ NIL
 3  do if $docID(p_1) = docID(p_2)$
 4        then Add($answer, docID(p_1)$)
 5             $p_1 \leftarrow next(p_1)$
 6             $p_2 \leftarrow next(p_2)$
 7        else if $docID(p_1) < docID(p_2)$
 8                then if $hasSkip(p_1)$ and ($docID(skip(p_1)) \leq docID(p_2)$)
 9                        then while $hasSkip(p_1)$ and ($docID(skip(p_1)) \leq docID(p_2)$)
10                               do $p_1 \leftarrow skip(p_1)$
11                        else $p_1 \leftarrow next(p_1)$
12                else if $hasSkip(p_2)$ and ($docID(skip(p_2)) \leq docID(p_1)$)
13                        then while $hasSkip(p_2)$ and ($docID(skip(p_2)) \leq docID(p_1)$)
14                               do $p_2 \leftarrow skip(p_2)$
15                        else $p_2 \leftarrow next(p_2)$
16  return $answer$
```

Example

- assume to be at 8 in both lists
- advance both pointers $\Rightarrow$ top: 16, bottom: 41
- 16 is smaller $\Rightarrow$ advance top pointer
  - check if taking the skip pointer still results in being $\leq 41 \Rightarrow$ if so, take it



2.6.2 Where do we place skips?

- trade-off: number of items skipped vs. frequency skips can be taken
  - more skips: fewer items skipped, but higher frequency
  - fewer skips: more items skipped, but lower frequency
- simple heuristic for placement: place $\sqrt{P}$ evenly placed skip pointers (where P: length of postings list)
  - ignores distribution of terms
  - easy for static indexes, harder for changing ones
- skip pointers used to help a lot, but with modern CPUs they do not help much anymore

2.7 Positional Index and Phrase Queries

2.7.1 Phrase Queries

- idea: support queries looking for exact phrases such as "stanford university"
- inverted index index with docIDs only is no longer sufficient

$\Rightarrow$ 2 alternatives: biword index and positional index

2.7.2 Biword Index

- index every consecutive pair of terms in the text as a phrase (e.g.: "I have a house" $\rightarrow$ "I have" and "have a" and "a house")
- each of these biwords is a vocabulary term
- longer phrase queries (>2 words): boolean AND queries (e.g.: "I have a" $\rightarrow$ "I have" AND "have a")
  - problem: results can contain false positives
  - solution: apply post filtering to filter for documents containing the exact phrase

Issues

- false positives
- index blowup (large vocabulary)

⇒ rarely used in practice

2.7.3 Positional Index

- more efficient alternative to biword indexes
- Postings lists in a

  nonpositional index: each posting is just a docID

  positional index: each posting is a docID **and a list of positions**: docID:(pos1, pos2, ...)
- can answer **Phrase Queries** and **Proximity Queries**

Processing a Phrase Query

1) obtain postings list of each query term
2) start with least frequent terms and further restrict candidates
3) intersection: similar as before BUT additionally
    - check that positions are compatible with query

2.8 Proximity Queries

- find documents featuring given terms in a given max. distance
- use positional index (not with a biword index)
- return the actual matching positions additional to the list of documents
- simple algorithm: look at cross-products of the positions of both terms for each document (extremely inefficient!)
- more efficient: **PositionalIntersection** - during intersection, check additionally the positions using a linear run-through strategy

```
PositionalIntersect($p_1, p_2, k$)
answer $\leftarrow \langle\,\rangle$
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then $l \leftarrow \langle\,\rangle$
           $pp_1 \leftarrow positions(p_1)$
           $pp_2 \leftarrow positions(p_2)$
           while $pp_1 \neq$ NIL
           do while $pp_2 \neq$ NIL
                 do if $|pos(pp_1) - pos(pp_2)| \leq k$
                       then Add($l, pos(pp_2)$)
                       else if $pos(pp_2) > pos(pp_1)$
                                then break
                    $pp_2 \leftarrow next(pp_2)$
                 while $l \neq \langle\,\rangle$ and $|l[0] - pos(pp_1)| > k$
                 do Delete($l[0]$)
                 for each $ps \in l$
                 do Add($answer, \langle docID(p_1), pos(pp_1), ps\rangle$)
                 $pp_1 \leftarrow next(pp_1)$
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
              then $p_1 \leftarrow next(p_1)$
              else $p_2 \leftarrow next(p_2)$
return answer
```

2.8.1 Combining Biword & Positional Indexes

- many biwords are very frequent (e.g. special names)
  $\Rightarrow$ Increased speed in postings intersection is very beneficial
  - include frequent biwords as vocabulary terms in the positional index $\Rightarrow$ no comparison of positions required
  - do all other phrases by positional intersection

CHAPTER 3

Dictionaries and Tolerant Retrieval

3.1 Search Structures for Dictionaries

- **dictionary**: datastructure for storing the term vocabulary
- **term vocabulary**: actual data
- dictionary maps from terms to associated data (i.e.: document frequency, pointer to postings list, etc.)
- two main concepts:
  - **hashing**
  - **(search) trees**
- relevant considerations for hashing vs. search trees
  1) number of terms
  2) number of terms fixed?
  3) relative term access frequency

3.1.1 Hashing

- hash each vocabulary term into an integer ⇒ row number in array
- at query time: hash query term and locate according entry in array

Advantages

- lookup is faster than in a tree (constant ($O(1)$))

Disadvantages

- impossible to find minor variants (resume vs. resumé)
- no prefix search (find terms starting with X)
- rehashing required for growing vocabulary
- possibly collision resolution required

3.1.2 Search Trees

- trees solve the prefix search problem
- simplest tree: **binary tree**
- search is slower than with hashing: $O(\log(|V|))$
  - $O(\log(|V|))$ holds only for **balanced trees**
  - rebalancing binary trees is expensive
  - $\Rightarrow$ use B-trees (automatic balancing during insertion)

B-Tree

- every internal node has a number of children (sorted) in the interval $[a, b]$ where $a, b \in \mathbb{N}_0^+$
- **adding an item**: similar as in binary tree BUT
  - if there is still a free spot in the children set of the node, the element is added
  - otherwise the children set of the node is split (and thus destroyed) evenly in two parts:
    1) median from elements in the node to split including the new element becomes new parent node of the splits
    2) median is inserted in the old parent node of the other elements
    3) new elements are added as children of the parent accordingly
  - $\Rightarrow$ might lead to recursive splits (and possibly a new root)

3.2 Wildcard Queries

- e.g. "mon*": find all documents containing any term beginning with "mon"
- **trailing wildcard query**: "mon*"
  - "*" at the end
  - easy with B-tree dictionary:
    1) retrieve all terms t in range: mon $\leq t <$ moo
    2) retrieve all documents featuring any of these terms (standard index)
- **leading wildcard query**: "*mon"
  - "*" at start $\Rightarrow$ find all docs containing a term ending with "mon"

- easy with **reverse B-tree**
  - additional tree storing all terms **spelled backwards**
- retrieve all terms t in range: nom $\leq t <$ non

- **general wildcard queries**
  - "*" in the middle of a term, e.g. "m*nchen"
  - easy solution: lookup "m*" and "*nchen" in the according B-tree and intersect two sets
  - BUT: very expensive
  - alternative: **permuterm index**

3.3 Permuterm Index

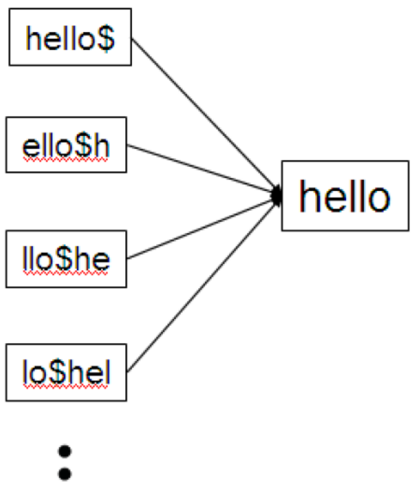
- special form of an inverted index
- introduce a special symbol $ to mark end of a term: ⇒ "hello" becomes "hello$"

3.3.1 Idea

- store various rotations of a term and map from them to the original term

⇒ store rotations in B-tree and let them point to the same postings list

- e.g.: for "hello" add the following versions to the B-tree:
  - hello$
  - ello$h
  - llo$he
  - lo$hel
  - o$hell
  - $hello



- **permuterm vocabulary**: set of rotated terms in permuterm index

3.3.2 Query Resolution

1) rotate query s.t. "*" is at the end (including \$ to mark end)
2) use B-tree search technique discussed earlier
   - $X \rightarrow X\$$
   - $X* \rightarrow \$X*$
   - $*X \rightarrow X\$*$
   - $*X* \rightarrow X*$
   - $X * Y \rightarrow Y\$X*$

- queries with multiple "*" might require a **pos-filter step**
  - "fi*mo*er" → er\$fi*
  - not all results will have "mo" in the middle ⇒ post filter check for "mo" is required

3.3.3 Problems of Permuterm Index

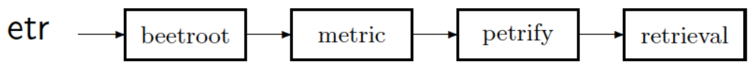
- roughly 4× as large (esp. the dictionary) as a regular B-tree

3.4 K-Gram Index

- *k*-gram: sequence of *k* characters
- special char \$ denotes beginning and end of a term
- more space efficient than permuterm index

3.4.1 Idea

- for each term:
  1) collect all of its k-grams
  2) store these in the dictionary
- maintain an inverted index from k-gram to all vocabulary terms containing the k-gram



3.4.2 Overall Setup

two indexes involved

1) **k-gram index**: finds terms based on a k-gram query
2) **term-doc index**: finds documents based on a term-query

3.4.3 Wildcard-Query Resolution

1) transform wildcard queries into boolean queries and run on k-gram index
    - assume bi-gram index
    - "mon*" → $m AND mo AND on
    - retrieves all terms from the index with the according requested prefix
    - BUT: also false positives like "moon"
2) post-filter terms to filter out false positives
3) lookup each term in the term-doc index and merge results (or do a combined OR query)

3.4.4 Problems of K-Gram Index

- potentially high number of boolean queries to be executed
    - becomes worse if we allow combinations: e.g. re*d AND fe*ri
- users hate to type
    - users will use wildcard queries a lot if allowed ⇒ very high cost for answering queries

3.5 Spelling Correction

- important for **tolerant retrieval**: handle imprecise queries which appear as a result of spelling errors, different writing ways, etc.
- two major concepts

3.5.1 Concepts

- two main use cases
    - correct documents being indexed
        - usually only done on documents scanned via optical recognition
        - general principle: **do not change docs**
    - correct user queries
- two main kinds of spelling correction
    - **isolated word spelling correction**
        - check each word on its own for misspelling

    - misses typos resulting in correctly spelled words: "fall form(!) the sky"

- **context-sensitive spelling correction**
    - considers surrounding words as well
    - catches error above

3.5.2 Isolated Word Spelling Correction (For Queries)

- two assumptions:
    - we have a list of "correct" words, containing the correct spelling of a word
    - we can compute a **distance** between a misspelled and correct word

List of Correct Words

- for the list of correct words, the vocabulary of the document collection can be used, BUT
    - if correct word is not contained, we might get strange query results
    - in general: computing distances against the whole vocabulary is too expensive

    ⇒ use some heuristic to pre-filter the vocabulary s.t. less distance computations are required
- alternatives:
    - weighted term vocabulary
    - standard dictionary (e.g. Duden)
    - weighted edit distances

Simple Algorithm for Isolated Word Spelling Correction

1) run over query and check for each word if it is in the list of correct words. If not, it is misspelled.
2) for each misspelled word: return correct word with smallest distance to wrong one
3) (return words with a distance $\leq$ threshold and show them as correction suggestion)

3.5.3 Context-Sensitive Spelling Correction

- option 1: **hit-based algorithm**
  1) for each query term, retrieve terms close to it (from dictionary) ⇒ result: sets of alternatives
  2) try all possible resulting phrases considering all possible combinations from the alternatives (with one word fixed at a time)
  3) assume that correct query as most hits

  ⇒ very expensive due to high amount of queries
- option 2:
  - instead of running the queries, measure their chance of being right according to frequency wrt. query history

  ⇒ choose the one which was queried most often in the past or yielded most results in the past

3.5.4 Distance Measures (for Strings)

Edit Distance

- distance between string s_1 and s_2 is the minimum number of basic operations required to transform s_1 into s_2
- example implementations:
  - **Levenshtein** basic operations: insert, delete and replace
  - **Damerau-Levenshtein** basic operations: additionally transposition

```
LevenshteinDistance($s_1, s_2$)
for $i \leftarrow 0$ to $|s_1|$
do $m[i,0] = i$
for $j \leftarrow 0$ to $|s_2|$
do $m[0,j] = j$
for $i \leftarrow 1$ to $|s_1|$
do for $j \leftarrow 1$ to $|s_2|$
    do if $s_1[i] = s_2[j]$
        then $m[i,j] = \min\{m[i\text{-}1,j]+1, m[i,j\text{-}1]+1, m[i\text{-}1,j\text{-}1]\}$
        else $m[i,j] = \min\{m[i\text{-}1,j]+1, m[i,j\text{-}1]+1, m[i\text{-}1,j\text{-}1]+1\}$
return $m[|s_1|, |s_2|]$
```

Operations: insert (cost 1), delete (cost 1), replace (cost 1), copy (cost 0)

- [**ToDo**: *add annotations in algo (which operation is which equation part*]
- how to read matrix (to get operations):
  1) start at bottom right
  2) at each step:
     a) look which sub cell is minimum (at tie: select any)

  b) note down the action corresponding to it
  c) go tht ecell belonging to the winning sub-cell

3) reverse order of actions

- note: for "replace or copy" you have to check if the cost increased
- left side of matrix: input, right side of matrix: output

cost of getting here from my upper left neighbor (copy or replace)	cost of getting here from my upper neighbor (delete)
cost of getting here from my left neighbor (insert)	the minimum of the three possible “movements”; the cheapest way of getting here

			s		n		o		w	
		0	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**4**
o		**1**	**1**	2	**2**	3	**2**	4	4	5
		1	2	**1**	**2**	**2**	3	**2**	**3**	**3**
s		**2**	**1**	2	**2**	3	**3**	**3**	**3**	4
		2	3	**1**	**2**	**2**	**3**	**3**	4	**3**
l		**3**	3	**2**	**2**	3	**3**	4	**4**	**4**
		3	4	**2**	3	**2**	**3**	**3**	**4**	**4**
o		**4**	4	**3**	**3**	**3**	**2**	4	4	5
		4	5	**3**	4	**3**	4	**2**	**3**	**3**

cost	operation	input	output
1	delete	o	*
0	(copy)	s	s
1	replace	l	n
0	(copy)	o	o
1	insert	*	w

- runtime: $O(|s_1| \cdot |s_2|)$
- **dynamic programming approach**
  - optimal substructures: optimal solution contais subsolutions
  - subsolutions overlap: are computed over and over again when using a brute-force algorithm
  - **subproblem for edit distance**: edit distance of two prefixes
  - **overlap for edit distance**: distances of prefixes are required 3 times
- variant: **weighted edit distance**
  - weight of an operations depends on characters involved
  - used for keyboard errors ("m" more likely to be mistyped as "n" than as "q")

k-Gram Overlap

- assume having a k-gram index
- given a query:
  1) for each term in the query, obtain all k-grams in the term
  2) use the k-gram index to retrieve "correct" words matching the k-grams of the query term
     - e.g.: return all terms having ≥ 2 k-grams in common with original term
     - e.g.: return all terms having a Jaccard distance (wrt. their k-grams) smaller than a threshold
     - in general: multiple options which require only one run over each k-gram entry relevant to the query

3.5.5 General Issues in Spelling Correction

- user interface
  - automatic vs. suggested correction
  - "did you mean" only works for one suggestion
  - trade-off: simple vs. powerful UI
- cost
  - spelling correction can be expensive
  - avoid running on every query?
  - guess: implementation on major search engines are efficient enough to run on every query

3.6 Phonetic Correction

- idea: look for "correct" words which **sound** similar
- especially useful for names: e.g. chebyshev vs tschebyscheff
- **concept**:
  - for a term: generate a "phonetic hash" s.t similar sounding terms hash to the same value

3.6.1 Algorithm (Class): Soundex

1) turn every term which should be indexed into a 4-char reduced from (build index mapping from these 4-char forms to originals)
2) do the same for the query terms
3) search for reduced forms of query terms in index
4) return result set as suggestions

in general NOT very useful in IR

CHAPTER 4

Index Construction

4.1 Hardware Basics

- access to data is much **faster in memory than on disk** (≈ factor 10)
- **disk seeks are idle time**: no data transfer during disk head movement
- optimizing transfer time from disk to memory: **one large chunk is faster than many small ones**
- **disk I/O is block based**: always read entire blocks instead of 1 byte
- **fault tolerance is expensive**: cheaper to use many regular machine than one fault tolerant one
- disk I/O is handled by the system bus (NOT CPU) ⇒ processor can be used to decompress data during reading

4.2 Problems of Default Approach

- standard process: process of parsing one doc at a time, inserting it into the according postings list and sort these at the very end
- not suitable for large collections as postings cannot be kept in main memory
- same algorithm on disk instead of memory is too slow due to too many disk seeks

⇒ need an external sorting algorithm

- crucial for this external algorithm: minimize number of disk seeks!

4.3 Blocked Sort-Based Indexing (BSBI)

4.3.1 Idea

- we only process as many (termID, docID) pairs as we can fit into main memory, sort these and write them to disk

4.3.2 Approach

1) segment collection of (termID, docID)-pairs into parts of equal size, called **blocks** (usually not done explicitly, but read using a buffer)
2) for each block:
   a) accumulate postings ((termID, docID)-pairs)
   b) sort postings in memory
   c) group postings by termID (and construct postings list)
   d) write result (inverted index for this block) to disk
3) merge intermediate results into one final index

```
BSBIndexConstruction()
1  $n \leftarrow 0$
2  while (all documents have not been processed)
3  do $n \leftarrow n + 1$
4      $block \leftarrow$ ParseNextBlock()
5      BSBI-Invert($block$)
6      WriteBlockToDisk($block, f_n$)
7  MergeBlocks($f_1, \ldots, f_n; f_{merged}$)
```

Remarks

- steps 2 a) and b) are called **inversion** steps

4.3.3 Merging of Intermediate Results

- open all intermediate results at the same time and maintain
  - one read buffer for each open file
  - one write buffer
- in each iteration:
  - select lowest termID not being processed yet
  - read the postings lists of this termID from the block buffers
  - merge these postings lists and write merged list to disk

4.3.4 Runtime

- $O(T \cdot \log(T))$ (T: upper bound of maximum number of (termID, docID) pairs of one block
- sorting is most dominant operation
- in practice: indexing time dominated by time of parsing and final merge

4.3.5 Problems

- we worked with a (term→termID)-mapping, i.e dictionary, which we kept always in memory
    - problematic as it grows and might exceed memory
- working with terms instead of termIDs makes intermediate files very large ⇒ process becomes slow

4.4 Single-Pass In-Memory Indexing (SPIMI

4.4.1 Ideas

- generate a separate dictionary for each block
- do not sort postings, but accumulate them on the fly (i.e. insert a posting at the right position when its found)

⇒ different inversion step than BSBI

4.4.2 Approach

1) for each block:
    a) for each (term,docID)-pair
        i. if term is unknown, add it to dictionary and create initial postings list with it
        ii. get postings list of term and add docID
            - if list was full, double its size before insertion
    b) sort (block-) dictionary alphabetically wrt. terms
    c) write index (of block) to disk
2) merge indexes of blocks into one index

```
SPIMI-INVERT(token_stream)
 1  output_file ← NEWFILE()
 2  dictionary ← NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6       then postings_list ← ADDTODICTIONARY(dictionary, term(token))
 7       else postings_list ← GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9       then postings_list ← DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

Merging of blocks is analogous to BSBI.

4.4.3 Remarks

- exponential size increase of list is efficient
- sorting step is required to allow final merge process as one linear pass
    - otherwise checking if a block contains data for a term requires linear time!
- compression makes SPIMI more efficient:
    - compression of terms
    - compression of postings

4.5 Distributed Indexing

4.5.1 Situation

- web-scale indexing requires a distributed cluster
- BUT: individual machines are fault-prone
    - assume non-fault-tolerant system with 1000 nodes, each having an uptime of 99.9%
    - up-time of total system: $0.999^{1000} = 0.367 \approx 37\%$

4.5.2 Idea

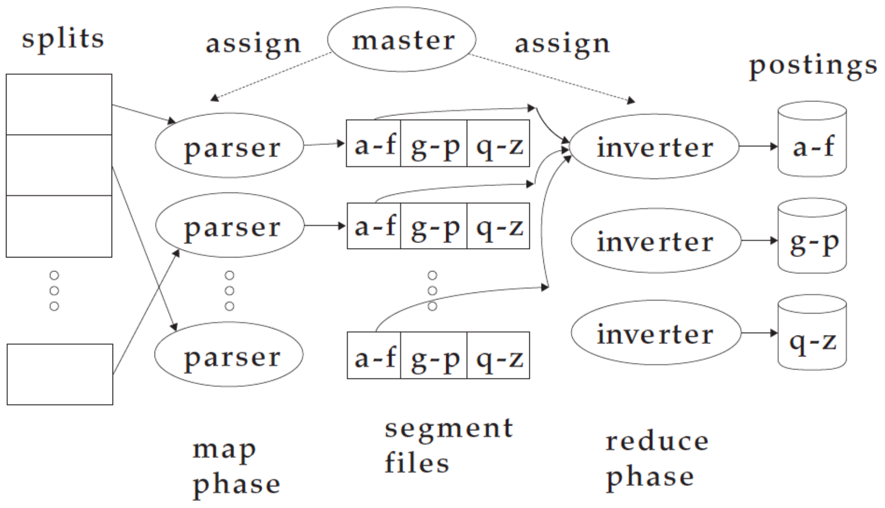
- break up indexing into sets of parallel tasks
- maintain a **master node/machine** directing jobs to machines (considered safe)
- two types of tasks
    - **parsing task** (performed on parsers)
    - **inverting task** (performed on inverters)
- to allow parallel tasks: break up document collection into **splits** (subset of documents)

4.5.3 Approach

1) break input document collection into splits
    - split size should neither be too small or too large
2) create one parsing task per split
3) master assigns split (task) to an idle parser machine
    - parser reads a document at a time and collects (term, docID)-pairs
    - pairs are written into j **term partitions** to disk
        - e.g.: one partition stores all terms starting with according letters $\Rightarrow$ for $j = 3$: a-f,g-p,q-z
        - other assignment strategies are possible too

4) once all parser tasks are done, one **inverter task for each term-partition** (j tasks) is scheduled

   - inverter collects all (term,docID)-pairs from one term-partition (consisting of multiple files (one per parser))
   - inverter sorts these pairs and writes them to postings lists



4.5.4 Remarks

- result of process is term-partitioned index
- search engines usually want document-partitioned indexes s.t. highly frequently queried docs can be put ins special indexes and queried first (tiered indexes)
- term-partitioned index can be transformed into doc-partitioned index
- whole process can be implemented as **MapReduce**:
  - robust and simple framework for distributed computing
  - removes overhead of writing the actual distribution code

4.6 Dynamic Indexing

4.6.1 Situation

- required when document collections are not static
  - usually docs are inserted, removed or altered

⇒ dictionary and postings lists have to be dynamically modified

4.6.2 Simple Approach

- maintain big main index on disk
- new docs go into smaller auxiliary index in memory
- query resolution: search across both indexes and merge results
- periodically merge auxiliary index into big one
- deletions:
  - add invalidation bit on deleted docs
  - on query resolution: filter docs returned by index and erase docs with that bit
- updates: delete and re-index

Issues of Simple Approach

- frequent merges
- poor overall runtime
  - n: size of auxiliary index
  - T: total number of postings
  - index construction requires $\left\lfloor \frac{T}{n} \right\rfloor$ merges
  - during a merge each posting which we have seen so far is touched
    - first merge: n postings involved
    - second merge $2n$ postings involved, etc.
  - total index construction time:

$$O\left(n + 2n + 3n + \ldots + \left\lfloor \frac{T}{n} \right\rfloor \cdot n\right) = O\left(\sum_{i=1}^{\left\lfloor \frac{T}{n} \right\rfloor} i \cdot n\right) = O\left(\frac{T^2}{n}\right) \tag{4.1}$$

- poor search performance during index merge

4.6.3 Logarithmic Merge Approach

- amortizes the cost of merging indexes over time ⇒ users see smaller effect on response time

Idea

- similar to simple approach, but use more indexes and change merge algorithm
- maintain a series of indexes, each twice as large as the previous one
  - Z-index: memory, l-indexes: disk
  - $|Z_0| = 2^0 \cdot n$ (main memory), $|l_0| = 2^0 \cdot n$, $|l_1| = 2^1 \cdot n$, etc.
- keep smallest index Z_0 in memory
- keep larger ones $(l_0, l_1, \ldots)$ on disk
- if Z_0 gets too large
  - if l_0 does not yet exist, write it to disk as l_0
  - if l_0 already exists, merge it with l_0 to create Z_1
  - try to write Z_1 to disk as l_1 (and continue process until its written)
- query resolution: run query against Z_0 and all l_i's present

```
LMergeAddToken(indexes, $Z_0$, token)
 1  $Z_0 \leftarrow$ Merge($Z_0$, {token})
 2  if $|Z_0| = n$
 3    then for $i \leftarrow 0$ to $\infty$
 4           do if $I_i \in$ indexes
 5                then $Z_{i+1} \leftarrow$ Merge($I_i$, $Z_i$)
 6                      ($Z_{i+1}$ is a temporary index on disk.)
 7                     indexes $\leftarrow$ indexes $- \{I_i\}$
 8                else $I_i \leftarrow Z_i$    ($Z_i$ becomes the permanent index $I_i$.)
 9                     indexes $\leftarrow$ indexes $\cup \{I_i\}$
10                     Break
11           $Z_0 \leftarrow \emptyset$

LogarithmicMerge()
 1  $Z_0 \leftarrow \emptyset$    ($Z_0$ is the in-memory index.)
 2  indexes $\leftarrow \emptyset$
 3  while true
 4  do LMergeAddToken(indexes, $Z_0$, getNextToken())
```

Runtime Remarks

- number of indexes bound by $O\left(\log\left(\frac{T}{n}\right)\right)$ (T: total number of postings read so far)

$\Rightarrow$ query resolution requires merging up to $O\left(\log\left(\frac{T}{n}\right)\right)$ result sets

- index construction takes $O\left(T \cdot \log\left(\frac{T}{n}\right)\right)$ time
  - each of the T postings is part of $O\left(\log\left(\frac{T}{n}\right)\right)$ merges

$\Rightarrow$ index construction is an order of magnitude more efficient than the simple approach

⇒ BUT: slower query processing

4.6.4 Dynamic Indexing at Large Search Engines

- often combination of
  - frequent incremental changes
  - rotation of large parts of the index which can be swapped in
  - sometimes a complete rebuild (unclear if Google can do something like that)
- note: building a positional index is the same problem, but with much larger intermediate data

CHAPTER 5

Index Compression

5.1 Why Compression (In General)

- less disk space ⇒ saves money
- keep more stuff in memory ⇒ increases speed (caching!)
- increase speed of data transfer from disk to RAM (reading compressed data and decompressing it is faster than reading uncompressed data)
- assumption: **fast decompression algorithm**

5.1.1 Why in IR?

- **dictionary**: make it small enough to keep in main memory
- **postings file**:
  - reduce required disk space

  ⇒ reduce time needed to read from disk
  - (large search engines keep large parts of the postings in memory)

5.1.2 Kinds of Compression

- **lossy compression**: discards information
  - jpeg, mp3, etc.
  - preprocessing steps such as removing stop words, applying porter stemmer, etc. can be viewed as such
- **lossless compression**: preserve all information
  - what we do in index compression

5.2 Statistical Properties

5.2.1 Heap's Law: Estimating the Term Vocabulary Size

- estimates vocabulary size M as a function of the collection size

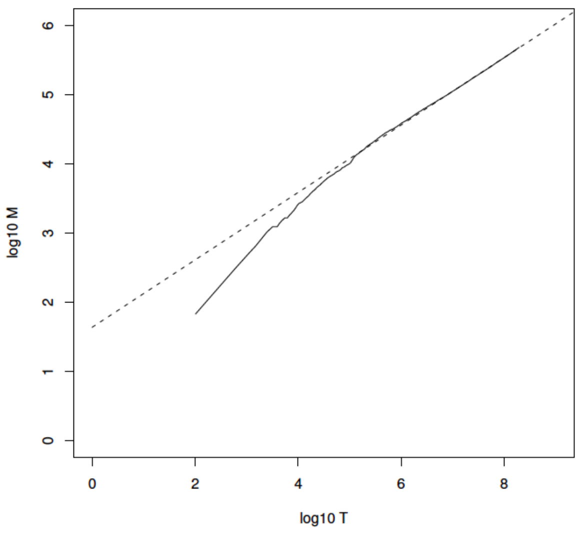
$$M = k \cdot T^b \tag{5.1}$$

  where

  - T: number of tokens in collection
  - k: parameter - usually $30 \leq k \leq 100$
  - b: parameter - usually $b \approx 0.5$

Remarks

- Heap's law is linear in log-log space
  - simplest possible relationship between collection size and vocabulary size in that space



- Shows vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1.
- Dashed lines (best least squares fit): $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit.
- $M = 10^{1.64} T^{0.49}$
  $\Rightarrow k = 10^{1.64} \approx 44$
  $\Rightarrow b = 0.49$.

- empirical law (not proven)
- **good fit** in general
- **implications**:
  - dictionary grows with more docs (no max. vocabulary size)

  $\Rightarrow$ compression is important

5.2.2 Zipf's Law: Modeling the Distribution of Terms

- intuition: in natural language we have a few very frequent terms and very many rare terms
- **collection frequency** of term t_i: number of times t_i appears in collection
- **assumption**: t_1 is the most frequent term, t_2 the next frequent one, etc. $\Rightarrow t_1 > t_2 > t_3 \dots$
- **law**: collection frequency cf_i of the i^{th} most common term is proportional to $\frac{1}{i}$

$$cf_i \propto \frac{1}{i} \tag{5.2}$$

- interpretation:
  - if most frequent term appears cf_1 times,
  - then the second most frequent one occurs $cf_2 = \frac{1}{2} \cdot cf_1$
  - then the third most frequent one occurs $cf_3 = \frac{1}{3} \cdot cf_1$, etc.

Zipf's Law as Power Law

1) version 1

$$cf_i = c \cdot i^k \tag{5.3}$$

2) version 2

$$\log(cf_i) = \log(c) + k \cdot \log(i) \tag{5.4}$$

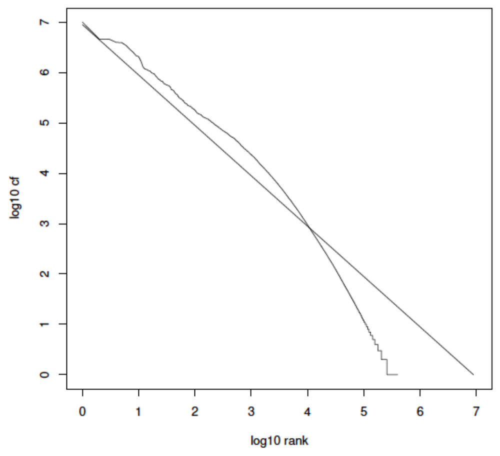
where

- $k = -1$
- c: constant

$\Rightarrow$ Zipf's law can be formulated as a power law

Properties

- overall **fit** is usually **not good**
- linear in log-log space



- key insight: few frequent terms, many rare terms

5.3 Dictionary Compression

- dictionary is small compared to postings

⇒ but: keeping it in main memory increases search performance drastically

5.3.1 Without Compression

- dictionary: fixed width array of
  - term (20 bytes)
  - frequency (4 bytes)
  - pointer to postings list (4 bytes)

	term	document frequency	pointer to postings list
	a	656,265	$\longrightarrow$
	aachen	65	$\longrightarrow$
	...	...	...
	zulu	221	$\longrightarrow$
space needed:	20 bytes	4 bytes	4 bytes

⇒ overall size: $|V| \cdot (20 + 4 + 4)$

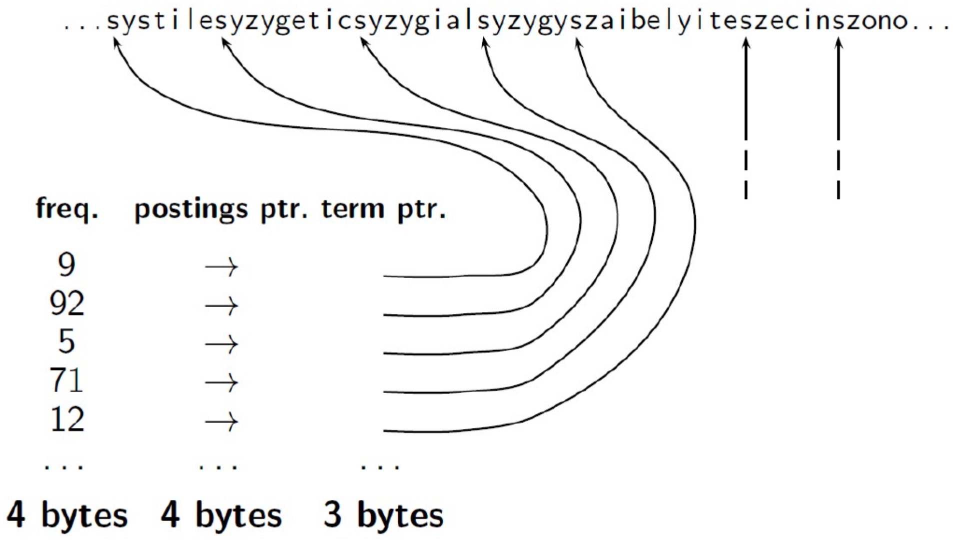
- **drawbacks**
  - wasting a lot of space for short terms
  - cannot handle terms with more than 20 chars

5.3.2 Dictionary as a String

- store all terms concatenated as one long string
- in addition store array with
  - frequency
  - postings list pointer
  - term pointer

⇒ term pointer marks beginning of term in string (next pointer marks end)

- terms are located in dictionary by binary search



- bytes per term pointer:
  - b_t: avg. bytes per term
  - then we need to be able to address $\log_2(b_t \cdot |V|)$ bits
  - with average of 8 bytes per term and 400k terms, we need to able to address 400k·8 positions in the string

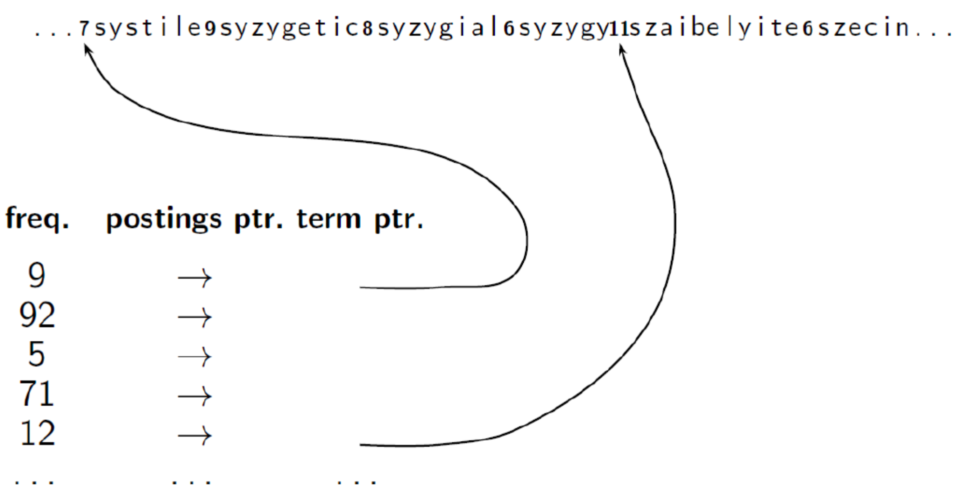
  ⇒ $\log_2(8 \cdot 400000) < 24\text{bits} = 3$ bytes for addressing

- overall dictionary size: $|V| \cdot (8 + 4 + 4 + 3)$

5.3.3 Blocked Storage

- similar to storing dictionary as string, but
  - reduce amount of term pointers by
    - grouping terms into blocks of size k
    - keeping only pointers to beginning of a block
  - to separate terms in a block: store each terms length as info before the actual term

⇒ eliminates $k - 1$ term pointers, but adds k bytes for storing the term lengths


. . . 7systile9syzygetic8syzygial6syzygy11szaibelyite6szecin. . .
freq.
postings ptr.
term ptr.
9
92
5
71
12
. . .
. . .
. . .

- higher $k \Rightarrow$ higher compression
  - BUT: **trade-off** between compression and term lookup speed

Term Lookup Details

terms are located in a two step approach

1) locate term's block by **binary search**
2) locate term in block by **linear search**

- average term lookup time in non-blocked vs. blocked storage
  - non-blocked: $\log_2(|V|)$
  - blocked: $\log_2\left(\frac{|V|}{B}\right) + \frac{B}{2}$ (B: block size)

5.3.4 Front Coding

- abuse common prefixes by storing them only once
- mark end of prefix with $*$ and replace it with $\diamond$ in subsequent terms

One block in blocked compression ($k = 4$) . . .

8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n

$\Downarrow$

. . . further compressed with front coding.

8 a u t o m a t $*$ a **1** $\diamond$ e **2** $\diamond$ i c **3** $\diamond$ i o n

5.3.5 Compression Efficiency

Summary (Reuters)

data structure	size in MB
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
dictionary, with blocking, $k = 4$	7.1
dictionary, with blocking & front coding	5.9

5.4 Postings File Compression

- postings file is much larger than the dictionary (factor 10+)
- if a posting consists of a docID, we would need $\log_2(|V|)$ $\Rightarrow$ we want to use a lot less

5.4.1 Key Idea

- key idea: store **gaps** instead of docIDs (after first docID) in postings list
  - for frequent terms, gaps are small (:= require less bits to store)
  - for rate terms, gaps are large (:= require more bits to store)

5.4.2 Variable Length Encoding Principle

- **aim**:
  - use more bits on rare terms to store gaps than for complete docIDs
  - use few bits to store gaps on frequent terms

$\Rightarrow$ few bits for small gaps and many bits for large gaps

5.4.3 Approach: Variable Byte Code (Form of VLE)

- byte based encoding

Approach

- use variable amount of bytes to encode a gap
- one byte is encoded as
  - first bit: **continuation bit**
  - last 7 bits: **payload** (encode the actual number or a part of it)
- **encoding**
  - take as many bytes as we need (assuming we use only the last 7 bits) and pretend these are all concatenated

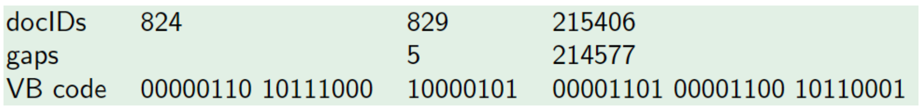
  - continuation bit is set to 1 on the **last** byte used and 0 otherwise

- **decoding**

  1) read sequence of bytes until continuation bit is set to 1 (including that byte)

  2) extract and concatenate 7-bit payloads into one number (in order read)

  3) convert binary to decimal number



docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Other Code Sizes (Than Byte)

- instead of bytes use other unit of alignments: 32 bits, 16 bits, 4bits (nibbles)
- larger units:
  - less amount of bit manipulation required ⇒ faster en-/decoding
  - BUT: less effective compression
- smaller units:
  - more amount of bit manipulation ⇒ slower en-/decoding
  - BUT: higher compression

5.4.4 Bitlevel Encoding

- higher compression possible when encoding on **bitlevel**
- easiest, but unefficient form: **unary code**

Unary Code

- represent n as n 1's ended by a 0
- e.g.: 3 ⇒ 1110, 10 ⇒ 11111111110

Gamma (γ) Code

- **encoding**: represent gap G as **pair of length and offset**
  - **offset** of G
    - G in binary form, but with leading 1 removed
    - (if there are 0s before the 1, these are removed too)
    - e.g.: $\mathit{offset}(13) = 101$ (as $\mathit{binary}(13) = 1101$)
  - **length** of G
    - length of **offset** (number of bits) encoded in unary code
    - e.g.: length of $\mathit{offset}(13)$ is 3 (bits) $\Rightarrow$ unary: 1110
  - **gamma** of G:
    - concatenation of **length** and **offset**
    - e.g.: $\gamma(13) = 1110, 101$ (, only for visual purposes)
- **decoding**
  1) read **length** until a 0 comes
  2) read number of bits defined by length read
  3) add 1 at the beginning of the read number
  4) transform to binary number

Length of Gamma Code

- length of gamma code is
  - always **odd**
  - in $O(\log_2(G))$ where G: size of gap
- proof
  - length (number of bits) of **offset**: $\lfloor \log_2(G) \rfloor$ bits
  - length (number of bits) of **length**: $\lfloor \log_2(G) \rfloor + 1$ bits

  $\Rightarrow$ entire length: $2 \cdot \lfloor \log_2(G) \rfloor + 1 \in O(\log_2(G))$

$\Rightarrow$ gamma codes are always of odd length

$\Rightarrow$ gamma codes are within a factor of 2 of optimal encoding length $\log_2(G)$ (assuming frequency of a gap G is proportional to $\log_2(G)$, which is only approximately true)

Properties of Gamma Code

- gamma code is **prefix-free** (as variable byte code)
  - a valid code word is not a prefix of any other valid code
  - no need for delimeters when concatenating codes (saves space)
- encoding is optimal with a factor of 2 or 3 (depending on assumptions)
  - 3-result is independent of the distribution of gaps ⇒ gamma-code is **universal**
- gamma code is **parameter-free** ⇒ easy to use

5.4.5 Remarks on Postings List Compression

- compressing and manipulating at bit-level can be slow as machines are made wot be fast on words: 8,16 and 32 bits
- variable byte encoding works with such words and thus potentially more efficient
- additionally: variable byte encoding is conceptually simpler at little additional cost

Chapter 6

On Scores, Weights and Vectors

6.1 Ranked Retrieval

- boolean model (matches or does not mach) has problems
  - goods for experts (who can express their needs precisely)
  - good for applications
  - BUT: bad for majority of users
    - unable (or not willing) to write good boolean queries
    - ⇒ most users do not want to read 1000s of results
  - **feast or famine problem**: queries have often too many (feast) or too few (famine) results
- with ranking, large result sets become handable (just show a part to the user)
- assumption: more relevant results are ranked higher than less relevant ones

6.1.1 Scoring as the Basis of Ranked Retrieval

- assign a score to each (query, document)-pair
  - measures how well query and document "match"
- sort documents for a query according to score

Features of a Good Score

- if no query term appears in the document, score should be 0
- the more frequent a query term is in a doc, the higher the score
- the more query terms occur in a doc, the higher the score

6.2 Parametric Zone Indexes

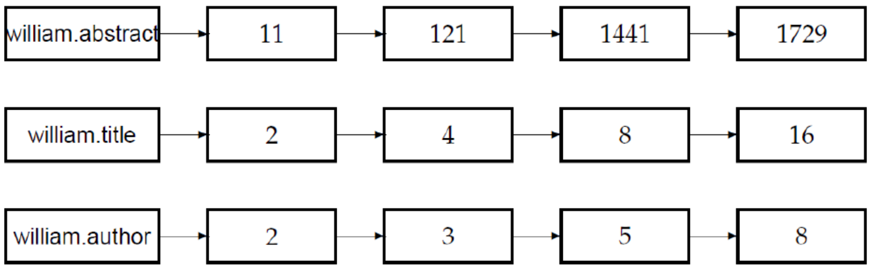
- allow to index and retrieve documents by metadata
- provide simple means for scoring

6.2.1 Context

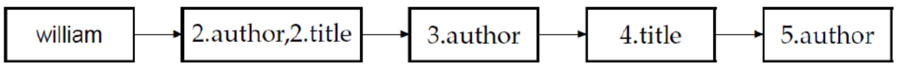
- docs usually have metadata (title, publication date, authors, etc.)
- metadata is commonly stored in **fields** (finite domain)
- **zone**: fields which contain any free text (e.g. abstract) (infinite domain)

6.2.2 Index Structure

- create one **parametric index** per field and one **zone index** per zone
  - ours standard inverted index can be seen as a zone index for the "text" field
- query resolution: merge results from indexes responsible for the fields being queried
  - query example: (author:"Martin", content:"Retrieval")
- practice: different parametric and zone indexes can be stored as one joint index



- Extend search algorithm to intersection of postings across required fields
- Zone can also be encoded in the postings



- second alternative reduces size of dictionary and allows for efficient **computation of scores using weighted zone scoring**

6.2.3 Weighted Zone Scoring

- way of computation a document score (wrt query) between 0 and 1
- consider set of docs with l zones/fields
- derive weights $g \in \mathbb{R}^l$ with

$$\sum_{i=1}^{l} g_i = 1 \tag{6.1}$$

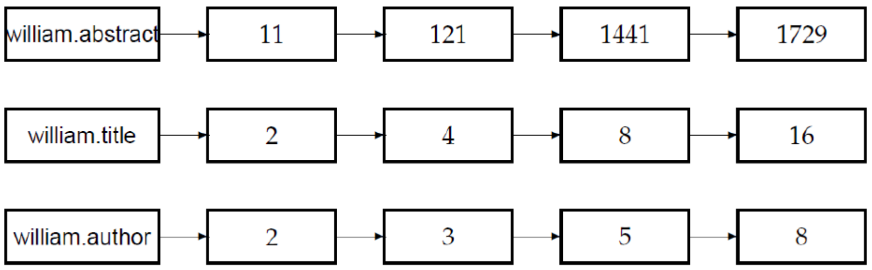
- **ranked boolean scoring**: weighted score $score(d, q)$ of a document

$$score(d,q) = \sum_{i=1}^{l} g_i \cdot s_i \tag{6.2}$$

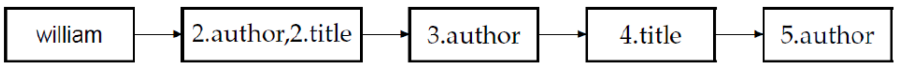
where

$$s_i = \begin{cases} 1 & \text{if doc is a match for zone } i \\ 0 & \text{else} \end{cases} \tag{6.3}$$

- weights s_i can be computed directly from an inverted index
- efficient computation can be done as part of the intersection operation:



- Extend search algorithm to intersection of postings across required fields
- Zone can also be encoded in the postings



Determining Weights g

- weights g_i have to be derived apriori
- e.g. via **machine learning**
- assumptions:
  - we have a training dataset of k entries (d_i, q_i, r_i) where
    - d_i := document i
    - q_i := query i
    - $r_i \in \{0, 1\}$: humanly created relevance flag (1 iff d_i is relevant to q_i)
- **squared error minimization**: find weight vector $g = (g_1, \ldots, g_l)$ which minimizes

$$\sum_{i=1}^{k} [score(d_i, q_i) - r_i]^2 = \sum_{i=1}^{k} \left[\left(\sum_{j=1}^{l} g_j \cdot s_j \right) - r_i \right]^2 \tag{6.4}$$

6.3 Jaccard Coefficient

- common measure to compute overlap of sets
- let A and B be two (non-empty) sets

$$JACCARD(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{6.5}$$

- interpretation
  - $JACCARD(A, B) \in [0, 1]$
  - $JACCARD(A, A) = 1$
  - $JACCARD(A, B) = 0$ if $A \cap B = \emptyset$
- score computation:
  - model both query and document as sets of words
  - compute JACCARD between query and document as a score
- problem of JACCARD
  - does not consider term frequency $\Rightarrow$ features of a good score function are violated

6.4 Term Frequency

6.4.1 Bag of Words Model

- represent document as **bag of words**: more frequent words are contained multiple times
- order of words is ignored
- use this model combined with special measures to compute scores

6.4.2 Term Frequency Measure

- idea: the more terms (from the query) a document contains (and the more often) the more relevant it is
- **term frequency** $tf_{t,d}$:= number of times t occurs in d
- BUT: raw term frequency is not good for computing scores as relevancy of a document does not increase proportionally with term frequency
  - e.g.: fix term t, doc d_1 with $tf_{t,d_1} = 10$ is more relevant than d_2 with $tf_{t,d_2} = 1$, BUT not $10\times$ more relevant

$\Rightarrow$ use **log frequency** of term

Log Frequency Weigh

- log frequency weight of term t in doc d is defined as

$$w_{t,d} = \begin{cases} 1 + \log_{10}\left(tf_{t,d}\right) & \text{if } tf_{t,d} > 0 \\ 0 & \text{else} \end{cases} \tag{6.6}$$

- examples: $tf_{t,d} \rightarrow w_{t,d}$: $0 \rightarrow 0$, $1 \rightarrow 1$, $2 \rightarrow 1.3$, $1000 \rightarrow 4$
- **score of (doc,query)-pair**: sum over terms t both in d and q

$$logScore(q, d) = \sum_{t \in q \cap d} \left(1 + \log_{10}\left(tf_{t,d}\right)\right) \tag{6.7}$$

6.4.3 Frequency in Document vs Frequency in Collection

- in addition to term frequency, **frequency of a term in the collection** is also used for ranking
  - a query term being **rare** in the collection is **more informative** than a frequent one $\Rightarrow$ very sure indicator of relevancy
  - a frequent query term in a document is a less strong indicator
- desired weights:
  - **high weights for rare terms**
  - **low** (but still positive) **weights for frequent terms**

$\Rightarrow$ document frequency df_t is used to express this in the score computation

- document frequency df_t: number of docs in collection where t occurs in

Quantity	Symbol	Definition
term frequency	$tf_{t,d}$	number of occurrences of t in d
document frequency	df_t	number of documents in the collection that t occurs in
collection frequency	cf_t	total number of occurrences of t in the collection

- document frequency is an **inverse measure of informativeness** of term t (i.e. low df := high informativeness)

6.4.4 Inverse Document Frequency (idf)

- inverse document frequency

$$idf_t = \log_{10}\left(\frac{N}{df_t}\right) \tag{6.8}$$

  where N: number of docs in collection

- *idf* is a **direct** measure of informativeness (i.e. high *idf* := high informativeness)

- other form: $idf^*_t \frac{\log_{10}(N)}{df_t}$ - reduces its effect

Effect on Ranking

- *idf* affects ranking of docs for queries with ≥ 2 terms
  - since weights of relevant terms are higher than the ones of not so relevant ones
- has little effect on ranking for one-term queries

Why *df* is better than *cf*

- *df*: document-level statistics
- *cf*: collection-level statistics
- document-level statistics are better to rank among docs (for scoring) than collection-level statistics
- see example: "try" has similar *cf* but higher *df* (and is certainly less informative than "insurance")

word	collection frequency	document frequency
insurance	10440	3997
try	10422	8760

6.4.5 Combined Ranking Scheme: *tf*-*idf* Weighting

- definition

$$tf\text{-}idf_{t,d} = \underbrace{\left(1 + \log_{10}\left(tf_{t,d}\right)\right)}_{tf_{t,d}} \cdot \underbrace{\log_{10}\left(\frac{N}{df_t}\right)}_{idf_{t,d}} \tag{6.9}$$

- interpretation
  - highest, when t occurs many times within a small number of docs
  - lower when t occurs fewer times in a doc OR occurs in many docs
  - lowest when t occurs in (almost) all docs
- best known weighting scheme in IR
- **score of a (doc,query)-pair**

$$tf\text{-}idfScore(q,d) = \sum_{t \in q} tf\text{-}idf_{t,d} \tag{6.10}$$

6.5 The Vector Space Model

- idea: represent both documents and queries as vectors

6.5.1 Documents as Vectors

- represent doc by vector of real values (one entry per vocabulary term)

$$d \in \mathbb{R}^{|V|} \tag{6.11}$$

  where V: vocabulary of collection
- one entry per term of vocabulary giving some weight for this (doc,term)-pair (e.g. *tf-idf*)

⇒ $|V|$-dim. real-valued vector space where terms are axes and docs are points
  - very high dimensional, but also sparse (most vector entries are 0)

6.5.2 Queries as Vectors

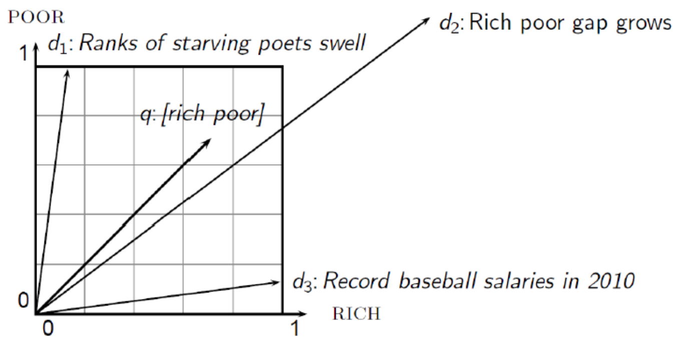
- same as for documents

Ranking Idea

- rank documents according to their **proximity** to the query
- proximity ≈ similarity
- proximity ≈ negative distance

6.5.3 Vector Space Similarity

- important to choose the right distance / similarity measure
- **Euclidean** distance is **bad**
  - **large for vectors of different length, although they are similar**
  - distance of q and d_2 is large although distribution of terms in q and d_2 are similar


POOR
1
d_1: Ranks of starving poets swell
d_2: Rich poor gap grows
q: [rich poor]
d_3: Record baseball salaries in 2010
0
0
1
RICH

- better: **angle between vectors**

⇒ **rank documents** according to the **angle between query and document** in **decreasing order**

- equivalent: rank according to **cosine(query, document)** in **increasing order**
  - works as cosine is monotonically decreasing wrt. the angle for the interval $[0, 180]$

Cosine Similarity

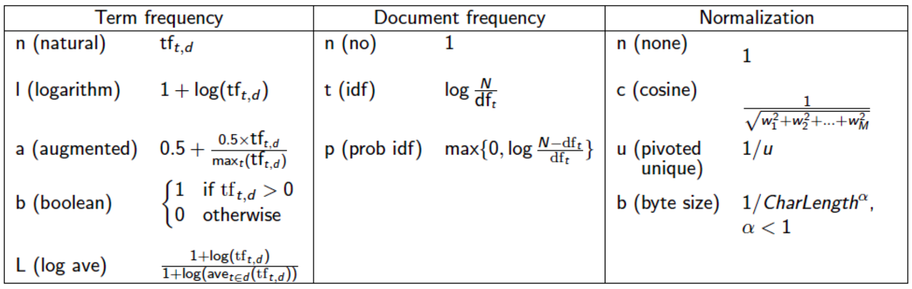
$$cos(q,d) = \frac{q \cdot d}{\parallel q \parallel_2 \cdot \parallel d \parallel_2} = \frac{\sum\limits_{i=1}^{|V|} q_i \cdot d_i}{\sqrt{\sum\limits_{i=1}^{|V|} q_i^2} \cdot \sqrt{\sum\limits_{i=1}^{|V|} d_i^2}} \tag{6.12}$$

where

- q_i: tf-idf weight of term t_i in query
- d_i: tf-idf weight of term t_i in document
- $\parallel d \parallel_2$: length of d, i.e. L_2 norm
- remarks
  - any vector q can be normalized by dividing it by its length, i.e. scale it to the unit interval s.t. $\parallel q \parallel_2 = 1.0$
  - cos is equivalent to the dot-product of normalized vectors

6.5.4 Overview of Schemes

- tf-idf is just a very special form of choosing weights for both the term-frequency (tf) and the document frequency (idf)
- in principle one can use different techniques and still apply the overall strategy
- points of decisions
  - weighting scheme for term-frequency
  - weighting scheme for document-frequency
  - normalization
  - distance measures
- best known combination: tf-idf + cosine normalization / distance measure



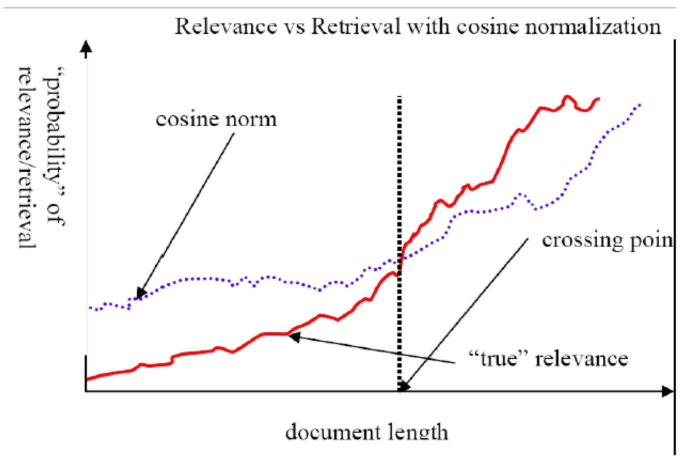
Term frequency		Document frequency		Normalization	
n (natural)	$\text{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\text{tf}_{t,d})$	t (idf)	$\log \frac{N}{\text{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2 + w_2^2 + \ldots + w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times \text{tf}_{t,d}}{\max_t(\text{tf}_{t,d})}$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

6.5.5 Pivot Normalization

- problem of cosine normalization: produces values too large for small documents and values too small for large documents

⇒ short documents have an unfair advantage

- adjust cosine normalization by linear adjustment: rotate cosine normalization curve counter clockwise at the pivot (crossing point) s.t. it matches the true relevance curves more closely

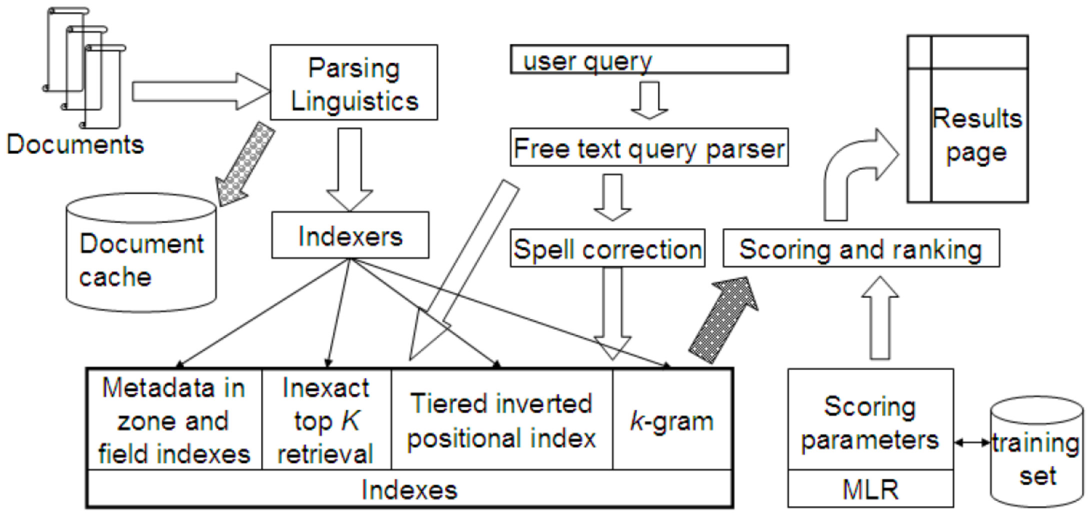


- implementation: instead of using Euclidean distance for length normalization of document vector, use a
  - larger one for documents shorter than at the pivot
  - smaller one for documents longer than at the pivot

CHAPTER 7

Search, Rank, Evaluate

7.1 Complete Search System Architecture



7.2 Why Rank?

- users want to look at few relevant results
- users want to find the requested information as fast as possible
- writing good queries (producing few results) is super hard in unranked retrieval

⇒ Large result set is reduced to small one

- getting top ranked page right is most important

7.3 Tiered Indexes

7.3.1 Idea

- used in top-k retrieval
- split up index into tiers
- each tier gets a threshold s.t. tier indexes only postings entries with a tf-value exceeding the threshold
- tier 1 has highest threshold and thresholds decrease
- query processing:
  - start at tier 1
  - if it produces less than k results, additionally consult tier 2, etc.

7.4 Computation of Top-K Ranking

- complete ranking is usually not required, but top-K is enough

7.4.1 General Complexity

- ranking has time complexity $O(N)$ (N: # of documents)
- essentially: solving the k-nearest neighbor problem for the query vector

$\Rightarrow$ no general solutions which are sublinear

7.4.2 Naive Approach

- compute scores for all N documents
- sort
- return top k

$\Rightarrow$ Extremely inefficient (esp. for large N)

- BUT: most documents of a similarity score of or close to 0! (we can speed things up)

7.4.3 Min-Heap Approach

- Idea: Run through all documents and keep the top k documents seen so far

Binary Min-Heap

- binary tree in which each node's value is less than the value of its children

Process

To process a new document d' with score s' do

- get current minimum h_m of heap ($O(1)$)
- if $s' \leq h_m$: skip to next document
- if $s' > h_m$: heap-delete-root ($O(\log(k))$)
- heap-add d' ($O(\log(k))$)

Runtime

- construction takes $O(N \log(k))$
- obtaining top-k takes $O(k \log(k))$

7.5 Heuristics Approach

- two main ideas
  1) reorder postings lists: instead of sorting wrt. docID, sort wrt. some measure of expected relevance
  2) use heuristics to prune search space

7.5.1 Non-docID Ordering of Postings Lists

- use a query-independent measure of goodness of a document
- e.g. page-rank $g(d)$ of page d
- order documents according to page rank: $g(d_1) > g(d_2)$...
- define **composite score** of a document

$$\text{net-score}(q, d) = g(d) + cos(q, d) \tag{7.1}$$

- supports **early termination** (no need to process complete postings lists):
  - suppose that
  - $g : D \to [0, 1]$
  - $g(d) < 0.1$ for doc d currently being processed
  - smallest top k score we have found so far is 1.2

  $\Rightarrow$ all subsequent scores will be < 1.1

  $\Rightarrow$ stop processing remainder of postings lists
- ordering of docs is consistent among terms

7.5.2 Document-at-a-time vs. Term-at-a-time

- document-at-a-time: complete computation of query-document similarity of document d_i before starting with d_{i+1}
  - e.g.: all previously seen approaches
- term-at-a-time: process postings list of a query term t_i before starting with query term t_{i+1} and accumulate information about documents
  - e.g.: next approach

7.5.3 Term-at-a-time Processing

- simple version:
  1) process postings list of first query term: create an accumulator for each docID you encounter
  2) process postings list of the second query term, etc.
- creating accumulators for all docs is infeasible for large collections
- for conjunctive search, we can even create an accumulator for a document only if all terms occur

7.5.4 Weight-Sorted Postings Lists

- order documents in postings according to weight (e.g. tf-idf - rarely done as hard to compress)
- stop after processing top k of postings lists
  - documents in top k are likely to occur early in these ordered lists $\Rightarrow$ early termination is unlikely to change top k
- BUT:
  - no consistent ordering of documents among terms
  - document-at-a-time processing is no longer possible

7.6 Evaluation

- user happiness is often the most relevant criterium
- usually user happiness := relevance of returned results
- standard methodology components:
  - benchmark document collection
    - docs should be representative of what we expect in reality
  - benchmark query (information need, i.e. a complete sentence of what is requested) collection
    - information needs should be representative of what we expect in reality

- evaluation measure (i.e. relevance measure)
  - relevance judgements should be tailored towards targeted user group

7.6.1 Query vs. Information Need

- two different information needs sometimes produce the same query
- BUT: results might be very good for one but really bad for the other information need
- i.e.: benchmarks should not have queries but information needs (out of which queries are constructed)

7.6.2 Accuracy (Measure)

- fraction of correctly classified documents

$$\text{accuracy} = \frac{TP + TN}{TP + FP + FN + TN}.$$

	Relevant	Nonrelevant
Retrieved	true positives (TP)	false positives (FP)
Not retrieved	false negatives (FN)	true negatives (TN)

- problem: in IR amount of true negatives is usually extremely high

⇒ returning 0 results gives a really large accuracy

⇒ accuracy is useless in the IR context

- returning something at the cost of some junk is better than returning nothing!

7.6.3 Precision & Recall (Measure)

- **precision**: fraction of retrieved documents that are relevant

$$\text{Precision} = \frac{\#(\text{relevant items retrieved})}{\#(\text{retrieved items})} = \mathbb{P}(\text{relevant}|\text{retrieved}) \tag{7.2}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7.3}$$

- **recall**: fraction of relevant documents that are retrieved

$$\text{Recall} = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = \mathbb{P}(\text{retrieved}|\text{relevant}) \tag{7.4}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{7.5}$$

Trade-Off

- increase recall by returning more documents (recall is a non-decreasing function of # docs retrieved)

$\Rightarrow$ messes up precision

- increase precision by only returning those documents which you are really sure about

$\Rightarrow$ messes up recall

F-Measure

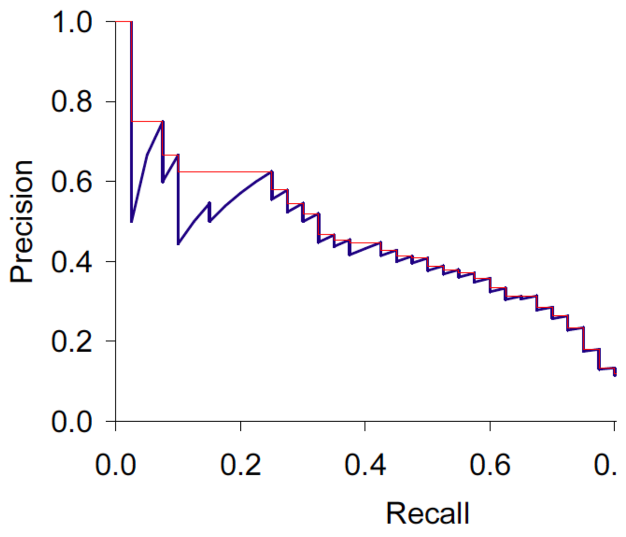
- combines recall and precision
$$F = \frac{(1+\beta^2) \cdot P \cdot R}{\beta^2 \cdot P + R} \tag{7.6}$$
where $\beta^2 \in [0, \infty]$
- for $\beta = 1$: **balanced F**, i.e. harmonic mean of P and R (equally weights P and R):
$$F_1 = \frac{2 \cdot P \cdot R}{P + R} \tag{7.7}$$
- $\beta < 1$: emphasizes precision
- $\beta > 1$: emphasizes recall
- harmonic mean is kind of a smooth minimum of P and R
  - minimum punishes really bad performance on either P or R
  - BUT is not smooth and hard to weight
  - e.g. arithmetic mean := 0.5 when returning NO results (too high)

Problem

- measures for unranked retrieval!

7.6.4 Precision-Recall Curve (Measure)

- turns set-based measures (P, R and F) into ranked based measures
- compute the set based measure for the top k of the results (where k is constantly increased)



- each point: result for the top k ranked results ($k = 1, 2, 3, \ldots$)
- saw-tooth shape: if the $(k+1)$th doc is nonrelevant, recall is the same as for top k, but precision drops
- easier to view: interpolation (red): take maximum of all future points
  - okay to do, as user is usually willing to look at more stuff if both precision and recall get better
- graph can be boiled down into table of 11 values: **11-point interpolated average precision**
  - take values of interpolated precision (red line) at values $0, 0.1, \ldots 1.0$ for all queries (one curve per query) and compute average values
  - this can be plotted as a curve

7.6.5 ROC Curve (Measure

- Receiver Operating Characteristics
- plots the true positives rate (recall / y-curve) against the false positives rate (FP/(FP + TN) / 1-specifity / x-curve)
- only interested in the area in the left corner
- the steeper the curve at the beginning, the better

7.7 Validity of Relevance Assessments

- for computing evaluation measures, we need relevance assessments by judges
- only usable if consistent among judges - otherwise there is no ground truth!
- way of measuring consistency: **kappa measure**

7.7.1 Kappa Measure

- measure of how much judges agree or disagree
- made for categorical judgments

$$\kappa = \frac{\mathbb{P}(A) - \mathbb{P}(E)}{1 - \mathbb{P}(E)} \tag{7.8}$$

  where

  - $\mathbb{P}(A)$ = proportion of times judges agree
  - $\mathbb{P}(E)$ = proportion of times judges are expected to agree by chance

- interpretation:
  - $\kappa = 1$: all judges agree always
  - $\kappa = 0$: judges agree only at the rate given by chance
  - $\kappa < 0$: judges agree worse than random
  - $\kappa \in \left[\frac{2}{3}, 1.0\right]$: acceptable agreement
- corrects for agreement by chance

7.7.2 Impact of Interjudge Disagreement

- impact of disagreement mostly on absolute performance numbers, but not on ranking of systems/algorithms

$\Rightarrow$ useful even if judges disagree a lot!

7.8 Evaluation at Large Search Engines

- recall is hard to measure on the web
- often one uses precision at top k or measures which reward getting the top ranked result right
- sometimes also other measures such as clickthrough rate

Chapter 8

Query Expansion and Relevance Feedback

- idea: increase recall ≈ number of documents returned relevant to user (loose definition)
- increasing recall wrt. loose definition might decrease recall wrt. true measure:
  - expand "jaguar" to "jaguar" AND "panthera"

  ⇒ might increase amount of relevant docs in top k, but eliminates some other relevant docs
- two options for improving recall:
  - **query expansion**: expand terms in query by alternative terms (global (usually analyzes whole corpus once) method)
  - **relevance feedback**: allow user to give interactive feedback on results to rerun search (local (only analyze query) method)

8.1 Relevance Feedback

8.1.1 Basic Idea

1) user issues short and simple query
2) search engine returns results
3) user marks some results are relevant (and possibly others as non-relevant)
4) search engines uses this information to compute a new representation of information need and returns results
5) new results are hopefully better

8.1.2 Centroid

- **centroid**: center of mass of a set of points

⇒ as docs are vectors (points) ⇒ compute centroid (i.e. representative instance) of documents

- definition
$$c(D) = \frac{1}{|D|} \sum_{d \in D} v(d) \tag{8.1}$$
where
  - D: set of documents
  - $v(d)$: vector representing document d

8.1.3 Rocchio (SMART) Algorithm

- move new query towards relevant documents and away from non-relevant documents
$$q_m = \alpha \cdot q_0 + \beta \cdot \underbrace{\frac{1}{|D_r|} \sum_{d_j \in D_r} d_j}_{c(D_r)} - \gamma \cdot \underbrace{\frac{1}{|D_{nr}|} \sum_{d_j \in D_{nr}} d_j}_{c(D_{nr})} \tag{8.2}$$

where

- q_m: modified query vector
- q_0: original query vector
- D_r / D_{nr}: sets of known relevant / nonrelevant documents
- α, β, γ: weights

Remarks

- Trade-off: α vs β/γ - the more judged documents, the higher should β/γ be as we have more reliable data to manipulate the original query.
- negative weights are set to 0 as they do not make sense in the vector space model (clipping)
- positive feedback is more valuable than negative feedback (implemented by $\beta > \gamma$)

8.1.4 Conditions for Improving Recall

1) user knows term vocabulary of collection well enough for initial query
   - Missmatch of searcher's/collection vocabulary (eg. cosmonaut vs astronaut)
2) relevant documents contain similar terms (s.t. moving in the vector space yields similar documents)
   - Several unrelated instantiations of the query (eg. [contradictory government policies])

8.1.5 Evaluation

- unfair evaluation: compare improvement of evaluation measure (on all documents) from initial query to adjusted query
    - will gain extremely high improvements
    - BUT: documents for which we know they are important (voted on by the user) give a lot of improvement
- fair evaluation: compare the improvement only on documents NOT judged by the user (called residual collection)
- one round of relevance feedback is often very useful, two are only marginally more useful

Remarks

- a true evaluation should also consider the time an approach takes
- alternative to relevance feedback: user changes and resubmits query
    - mostly used as most users are reluctant to classify documents as relevant

8.1.6 Problems

- creates long modified queries (expensive to process)
- users often do not want to provide explicit feedback
- interpretability: hard to understand why a document was retrieved for the adapted query

8.2 Pseudo-Relevance Feedback

Idea

1) execute initial query
2) assume top k result to be relevant
3) do relevance feedback based on this assumption

Success

- works well on average
- but can be very wrong on some querys
    - reason: query drift - when apply PsRF to often, the final query is too far off the original one

8.3 Query Expansion

8.3.1 Basic Idea

- increases recall
- global query expansion: query is modified based on some global resource (being query independent, e.g. a dictionary of synonyms)

8.3.2 Global Resources

- thesaurus: database which collections (near-)synonyms
    - manual thesaurus (derived by experts)
    - automatically derived (e.g. based on co-occurrences of words in the document collection)
- query-equivalence based on query log mining

8.3.3 Thesaurus Based Query Expansion

- for each term t in the query, expand the query with the according results for t of the thesaurus
- increases recall, but might decrease precision (terms which can have different meanings)
- mostly used on specialized engines for science, etc.

8.3.4 Automatic Thesaurus Generation

- analyze distribution of words (and their similarity) in documents
- two ways to define similarity between words
    - co-occurrence: similar, if they co-occur with similar words (e.g. car $\approx$ motorcycle, as they both often occur together with road, etc.)
    - grammatical relation: similar if they occur in a given grammatical relation with the same words (e.g. you harvest, eat, prepare, etc. apples)
- co-occurrence is more robust, but grammatical relation more accurate

8.3.5 Use in Search Engines

- main source of expansion: query logs
    - user adjusting their queries
    - user clicking on two links with different wording, etc.

Chapter 9

Structured Retrieval

9.1 Relational Databases vs Unstructured IR

	RDB search	unstructured IR
objects	records	unstructured docs
main data structure	table	inverted index
model	relational model	vector space & others
queries	SQL	free text queries

9.2 Structured Retrieval Setting

- queries: structured or unstructured
- documents: structured (e.g. named entity tags, other tags, etc.)

9.2.1 Problems of RDBs in Structured Retrieval

- returns a potentially large number of documents which are not ranked at all
- difficult for users to precisely state structure contraints
- users may be completely unfamiliar with structured search and according search interfaces (or reluctant to use them)

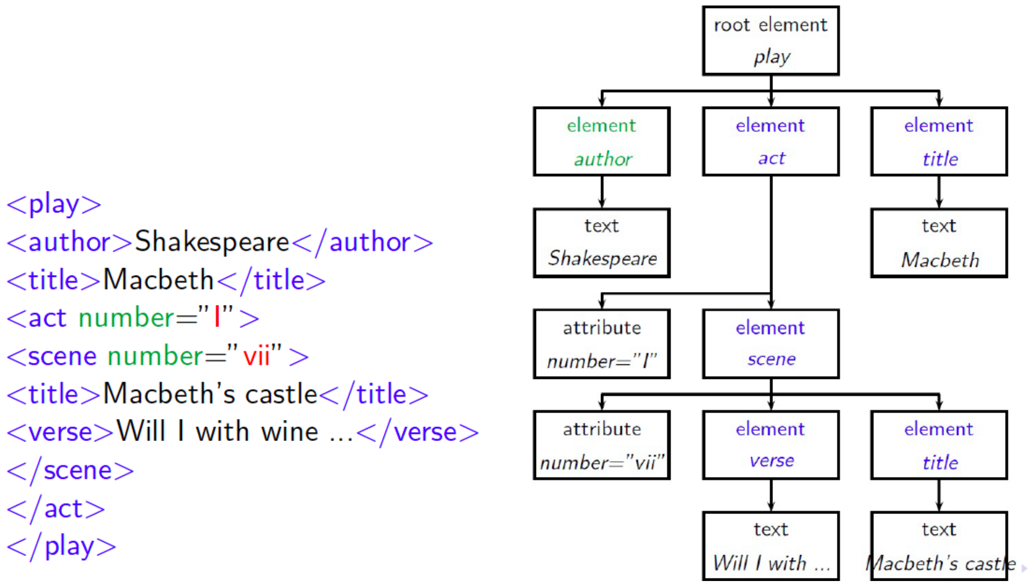
⇒ solution: adapt ranked retrieval to structured documents

9.2.2 XML

9.2.3 XML Basics

- ordered, labeled tree
- each node is an XML element

- elements can have one or multiple attributes (number)
- attributes can have values (vii)
- elements can have child elements (title, verse, etc.)



```
<play>
<author>Shakespeare</author>
<title>Macbeth</title>
<act number="I">
<scene number="vii">
<title>Macbeth's castle</title>
<verse>Will I with wine ...</verse>
</scene>
</act>
</play>
```

9.2.4 Notation

- XML DOM: document object model
  - standard or accessing and processing XML docs
  - elements, attributes and text are represented as nodes in a tree
  - API allows processing doc by starting at the root and descending down the tree
- XPath: standard for enumerating paths (also called contexts) in an XML document collection
- Schema: puts constraints on the structure of allowable XML documents

9.2.5 Challenges

First Challenge

return parts of documents and not complete ones (usually done by IR systems)

- user usually only interested in a specific part of the document
- solution: structured document retrieval principle
  - "system should always return the most specific part of a document answering a query"
- hard to implement, as if multiple levels satisfy the query, it can be hard to decide which one to return (highest one is usually chosen)

Second Challenge

how to structure the index of an XML document, i.e which document / indexing unit to use?

1) group nodes into non-overlapping pseudodocuments
    - problem: pseudodocuments make few sense to a user as they are non-coherent units
2) top-down
    - use one of the largest elements as the indexing unit
    - when answering a query, search in the index for the best fitting document and do post-processing to find its relevant sub-element
    - problem: relevance of document often not a good predictor of relevance of sub-elements, i.e. best sub-element might not be found
3) bottom-up
    - use leaves as indexing unit
    - when answering a query, search in the index for the best fitting leaf and do post-processing to extend it to larger units
    - problem: same as for top-down
4) index all elements
    - problem 1: highly redundant search results
        - since one element is often contained in a another (nested element), the information of the included one is duplicate
    - problem 2: some elements are often useless as search results

Third Challenge

redundancy caused by nested elements

- common solution: do not allow all elements to be retrieved
- restriction strategies:
    - discard all elements
    - discard all element types that users to not look at
    - discard all elements deemed unimportant by experts
    - only keep items which have been selected by an expert

    ⇒ most approaches only reduce amount of nested elements
- further solutions:
    - post-processing: remove nested elements in post-processing
    - collapse and highlight: collapse nested elements in the result list and use highlighting of query terms to improve user friendliness
        - gain 1: scanning a medium sized document can now be done a lot quicker (in comp. to no highlighting)

  – gain 2: user has information about context of search results

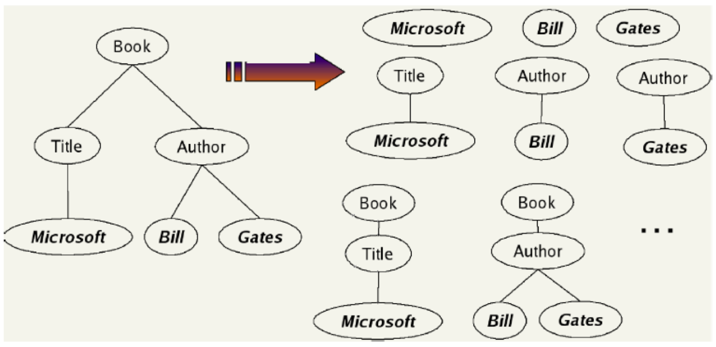
Fourth Challenge

distinguish different contexts of a term when computing term statistics

- a term under one node might be completely unrelated to the same term (having a different meaning for example) in another node
- solution: compute statistics not for terms but for (context,term) pairs
  - problem: results in sparse data, as many (context, term) pairs occur rarely
  - ⇒ compromise: consider the parent node of the term as the context and the whole path
- implementation: lexicalized subtrees

9.2.6 Lexicalized Subtree Vector Space Model

- idea: each dimension of the vector space encodes a (term, context) pair
- approach: map xml documents to lexicalized subtrees
  - take each leaf (text node) and break it into multiple nodes - one for each word
  - create one vector space dimension for each lexicalized subtree occurring in any document and containing at least one vocabulary term



- represent both documents and queries in this vector space

9.2.7 Curse of Dimensionality

- the more dimensions, the more accurate the search results (possibly), but also higher time and space requirements
- extreme cases:
  - restrict dimensions to raw vocabulary terms (no contexts) ⇒ standard vector space model (i.e. bad results in structured IR)
  - create one dimension for each lexicalized subtree occurring in the collection ⇒ dimensionality becomes too high to allow efficient computations
- compromise: index all paths that end in a single vocabulary term (i.e. all (context, term) pairs)

- such a (context, term)-pair $\langle c, t\rangle$ is called **structural term**

9.2.8 Context Resemblance

- simple measure of the similarity of a path c_q in a query and a path c_d in a document

$$C_R(c_q, c_d) = \begin{cases} \dfrac{1+|c_q|}{1+|c_d|} & \text{if } c_q \text{ matches } c_d \\ 0 & \text{else} \end{cases} \tag{9.1}$$

  where

  - $|c_q|$ and $|c_d|$: number of nodes in the according path
  - c_q matches c_d iff we can transform c_q into c_d by inserting additional nodes

- interpretation:
  - $C_R(c_q, c_d) = 1$ if c_q and c_d are identical
  - $C_R(c_q, c_d) = 0$ if they are maximally different

9.2.9 Document Similarity Measure

- using a variant of the cosine measure: *SimNoMerge*

$$\mathit{SimNoMerge}(q, d) = \sum_{c_k \in B} \sum_{c_l \in B} C_R(c_k, c_l) \sum_{t \in V} w(q, t, c_k) \frac{w(d, t, c_l)}{\sqrt{\sum_{c \in B, t' \in V} w^2(d, t', c)}} \tag{9.2}$$

  where

  - V: vocabulary of non-structural terms
  - B: set of XML contexts
  - $w(q, t, c)$ / $w(d, t, c)$: weights of term t in XML context c in query q / document d (e.g. tf-idf)

- *SimNoMerge* is not a true cosine measure as it can be > 1

9.2.10 Problems of XML

- designed for document content
- structure must be a tree
- hard to merge subtrees

9.2.11 Requirements of a Better Model

- support polyfaceted data sources
- support easy integration
- support data distribution
- easy integration with web standards

9.3 RDF (Resource Description Framework)

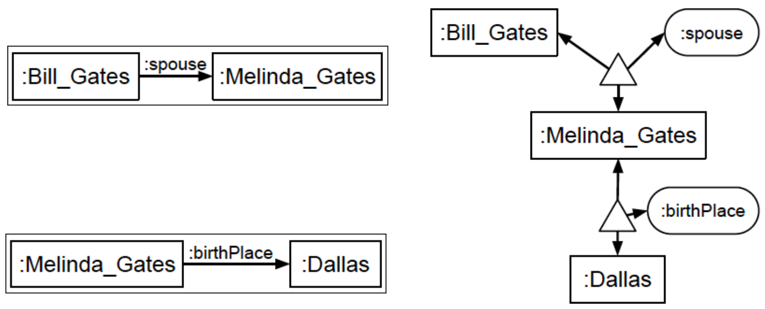
- basic components:
  - URI/IRIs: assigns a unique reference to resources (objects)
  - literals: describe data values (e.g. a number)
  - blank nodes: resource with a local but no global identity
- RDF graph: set of triples
- triple consists of
  - subject: URI or blank node
  - predicate: URI
  - object: URI, blank node or literal
- literals can have datatypes or language declarations (only on untyped literals)
  - e.g. "3.0"8sd:double
  - e.g. "Berlin"@de
- literals are mapped to a range space for interpretation (using the datatype) (e.g. "3.14000" and "3.14" both become 3.14)

9.3.1 RDF IR

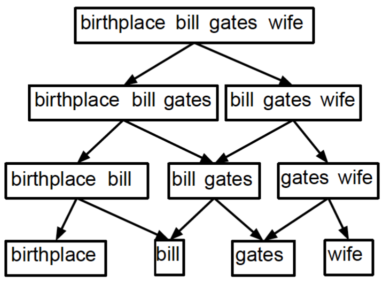
- goal: given a keyword query q, return the set of subgraphs which answer the query
- challenges:
  - semantic mismatch (person vs. human)
  - schema mismatch (father vs. parent AND male)
  - expressiveness
  - verbalization of results
  - complexity of graph matching is NP-hard

9.3.2 Query Answering

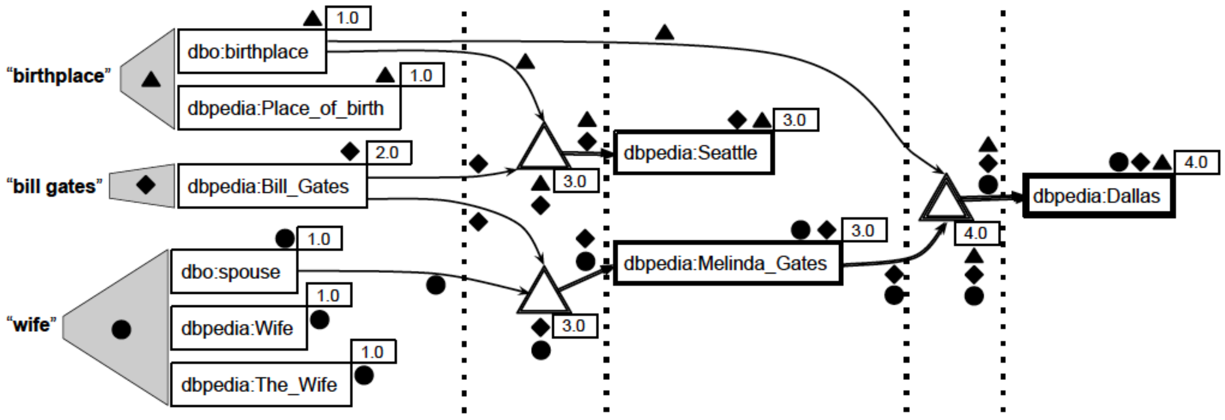
1) reification: transform each triple (s, p, o) of the graph into three edges $(f, s), (f, p), (f, o)$



2) query preprocessing: decompose query into tree



3) run color spreading



- assign colors to each keyword (sequence)
- colors mapped to overlapping keyword sequences exclude each other
- use sequence length as an explanation weight
- run across graph
  - mark nodes answering a keyword with the according color
  - mark nodes with the according weight if they are part of an answer sequence
  - terminate if a node fully explains the query

Chapter 10

Probabilistic Retrieval

10.1 Idea

- IR systems have an **uncertain** understanding of the user query and make an **uncertain** guess whether a document is relevant or not
- probability theory provides a principled foundation for such **reasoning under uncertainty**

10.2 Probabilistic vs Other Models

- boolean model
  - prob. models support ranking (by sorting by the prob.) $\Rightarrow$ better than boolean model
- vector space model
  - vector space model also supports ranking BUT
  - documents are ranked according to similarity to query
  - notion of similarity does not translate directly to "is this a good document to return or not?"

  $\Rightarrow$ most similar document can be highly relevant or completely nonrelevant

  $\Rightarrow$ probability theory allows for a cleaner formalization of "give relevant documents to the user"

10.3 Document Ranking Problem

- **ranked retrieval setup**: given a collection of documents, the user issues a query and an ordered list of documents is returned
- **relevance definition**

- binary notion of relevance: binary random variable

$$R_{d,q} = \begin{cases} 1 & \text{if document } d \text{ is relevant wrt query } q \\ 0 & \text{else} \end{cases} \tag{10.1}$$

- probabilistic ranking orders documents decreasingly wrt. their estimated probability of relevance wrt. the query
  - i.e. order by documents by $\mathbb{P}(R = 1|q, d)$
- **assumption**: relevance of each document is independent of the relevance of other documents
  - can be a horrible thing: returning a list of near duplicates is horrible!

10.4 Probability Ranking Principle

- rank retrieved documents decreasingly wrt. their probability of relevance (estimated from the given data)
  ⇒ effectiveness of the system is the best it can be (using the given data)

10.5 Requirements if Probabilistic IR systems

- need a reliable estimate of how terms in documents contribute to relevance

⇒ have measurable statistics (term frequency, document frequency, etc.) that affect the document relevance

- these statistics have to be combined to estimate the probability $\mathbb{P}(R|q, d)$ of document relevance

10.6 Binary Independence Model (BIM)

10.6.1 Main Assumptions

- **binary model**: document and queries are represented as binary term incidence vectors $x \in \{0, 1\}^{|V|}$
  - $x_t = 1$ if term t occurs in the corresponding document / query and $x_t = 0$ else
- **independence of terms**: no association between terms
  - of course wrong, but works in practice
  - $\approx$ Naive Bays assumption

10.6.2 Foundation

- $\mathbb{P}(R|d', q')$ modeled using term incidence vectors as $\mathbb{P}(R|x, q)$, where $x, q \in \{0,1\}^{|V|}$

$$\begin{aligned} \mathbb{P}(R=1|x,q) &\overset{\text{Bayes}}{=} \frac{\mathbb{P}(x|R=1,q)\cdot\mathbb{P}(R=1|q)}{\mathbb{P}(x|q)} \\ \mathbb{P}(R=0|x,q) &\overset{\text{Bayes}}{=} \frac{\mathbb{P}(x|R=0,q)\cdot\mathbb{P}(R=0|q)}{\mathbb{P}(x|q)} \end{aligned} \tag{10.2}$$

  where

  - $\mathbb{P}(x|R=1,q)$ / $\mathbb{P}(x|R=0,q)$: probability that if a relevant / non-relevant document is retrieved, its representation is x
  - $\mathbb{P}(R=1|q)$ / $\mathbb{P}(R=0|q)$: prior probability of retrieving a relevant / non relevant document for a query q (without specific knowledge about the document)

- documents are either relevant or not ⇒

$$\mathbb{P}(R=1|x,q) + \mathbb{P}(R=0|x,q) = 1 \tag{10.3}$$

10.6.3 Deriving a Ranking Function

- obvious ranking: rank documents according to $\mathbb{P}(R=1|x,q)$ in decreasing order
- easier: rank documents according to their **odds of relevance** $O(R|x,q)$
  - odds of relevance is monotonic with the probability of relevance ⇒ ranking does not change
  - a lot easier to compute in practice

$$\begin{aligned} O(R|x,q) &= \frac{\mathbb{P}(R=1|x,q)}{1-\mathbb{P}(R=1|x,q)} \\ &= \frac{\mathbb{P}(R=1|x,q)}{\mathbb{P}(R=0|x,q)} \\ &= \frac{\left(\frac{\mathbb{P}(x|R=1,q)\cdot\mathbb{P}(R=1|q)}{\mathbb{P}(x|q)}\right)}{\left(\frac{\mathbb{P}(x|R=0,q)\cdot\mathbb{P}(R=0|q)}{\mathbb{P}(x|q)}\right)} \\ &= \underbrace{\frac{\mathbb{P}(R=1|q)}{\mathbb{P}(R=0|q)}}_{O(R|q)}\cdot\frac{\mathbb{P}(x|R=1,q)}{\mathbb{P}(x|R=0,q)} \end{aligned} \tag{10.4}$$

  - $O(R|q)$: constant for a given query and thus can be ignored for computing the ranking

- using the **Naive Bayes assumption** (terms occur independently of each other in a document) we have

$$\frac{\mathbb{P}(x|R=1,q)}{\mathbb{P}(x|R=0,q)} = \prod_{t=1}^{|V|}\frac{\mathbb{P}(x_t|R=1,q)}{\mathbb{P}(x_t|R=0,q)} \tag{10.5}$$

- and thus

$$O(R|x,q) = O(R|q)\cdot\prod_{t=1}^{|V|}\frac{\mathbb{P}(x_t|R=1,q)}{\mathbb{P}(x_t|R=0,q)} \tag{10.6}$$

- since $x_t \in \{0,1\}$ we can separate the terms:

$$O(R|x,q) = O(R|q) \cdot \prod_{t:x_t=1} \frac{\mathbb{P}(x_t=1|R=1,q)}{\mathbb{P}(x_t=1|R=0,q)} \cdot \prod_{t:x_t=0} \frac{\mathbb{P}(x_t=0|R=1,q)}{\mathbb{P}(x_t=0|R=0,q)} \tag{10.7}$$

- for simplicity of notation we define
  - $p_t = \mathbb{P}(x_t=1|R=1,q)$ (probability that term t occurs in a document relevant for the query)
  - $u_t = \mathbb{P}(x_t=1|R=0,q)$ (probability that term t occurs in a document non-relevant for the query)
  - can be displayed as a table

	document	relevant ($R=1$)	nonrelevant ($R=0$)
Term present	$x_t=1$	p_t	u_t
Term absent	$x_t=0$	$1-p_t$	$1-u_t$

- additional **simplifying assumption**: terms not occurring in the query are equally likely to occur in relevant and nonrelevant documents
  - if $q_t = 0 \Rightarrow p_t = u_t$, i.e. those terms can be ignored in the equation as their ratio is 1

$$O(R|x,q) = O(R|q) \cdot \prod_{t:x_t=q_t=1} \frac{p_t}{u_t} \cdot \prod_{t:x_t=0,q_t=1} \frac{1-p_t}{1-u_t} \tag{10.8}$$

- to get rid of the right product, we include the query terms found in the document into it (i.e. dropping the condition $x_t = 0$
  - in order to not change the equation, we have to divide by them and we do so in the left product

$$O(R|x,q) = O(R|q) \cdot \prod_{t:x_t=q_t=1} \frac{p_t \cdot (1-u_t)}{u_t \cdot (1-p_t)} \cdot \underbrace{\prod_{t:q_t=1} \frac{1-p_t}{1-u_t}}_{\text{constant}} \tag{10.9}$$

  - the right product is now over all query terms in the query and thus a constant for a given query and thus can be ignored
- the only important quantity for ranking is the left product!
- for computational reasons, we do not use this quantity directly, but its logarithm (which is monotonic and thus produces the same ranking), called the **Retrieval Status Value (RSV)**

$$RSV_d = \log\left(\prod_{t:x_t=q_t=1} \frac{p_t \cdot (1-u_t)}{u_t \cdot (1-p_t)}\right) = \sum_{t:x_t=q_t=1} \underbrace{\log\left(\frac{p_t \cdot (1-u_t)}{u_t \cdot (1-p_t)}\right)}_{c_t} \tag{10.10}$$

- c_t is called the **log-odds ratio**

$$c_t = \log\left(\frac{p_t \cdot (1-u_t)}{u_t \cdot (1-p_t)}\right) \tag{10.11}$$

- ratio of odds of t appearing if the document is relevant $(\frac{p_t}{1-p_t})$ and the odds of appearing if the document is not relevant $(\frac{u_t}{1-u_t})$
- interpretation
  - $c_t = 0$: equal odds to appear in relevant and non-relevant docs
  - $c_t > 0$: higher odds to appear in relevant docs
  - $c_t < 0$: higher odds to appear in non-relevant docs
- note: c_t is essentially a **term weight** $\Rightarrow$ BIM and vector space model can be implemented identically but use different term weights

10.6.4 Computing Probability Estimates in Theory

- for each term t in a query, we estimate c_t by creating a contingency table of counts of documents in the collection (df_t : number of docs containing term t)

	documents	relevant	nonrelevant	Total
Term present	$x_t = 1$	s	$\mathrm{df}_t - s$	df_t
Term absent	$x_t = 0$	$S - s$	$(N - \mathrm{df}_t) - (S - s)$	$N - \mathrm{df}_t$
	Total	S	$N - S$	N

$$p_t = s/S$$

$$u_t = (\mathrm{df}_t - s)/(N - S)$$

$$c_t = K(N, \mathrm{df}_t, S, s) = \log \frac{s/(S-s)}{(\mathrm{df}_t - s)/((N - \mathrm{df}_t) - (S - s))}$$

- this way of estimation is **maximum likelihood estimation** as the value above makes the observed data maximum likely
- if any of the counts is 0, c_t is not well defined
- solution: apply smoothing, e.g. add $\frac{1}{2}$ to each quantity in the inner part of the table

10.6.5 Computing Probability Estimates in Practice

Estimating u_t

- **assumption**: relevant documents are a very small percentage of the collection

$\Rightarrow$ approximate statistics for non-relevant documents by statistics from the whole collection

$\Rightarrow$ $u_t = \frac{df_t}{N}$ and

$$\log\left(\frac{1-u_t}{u_t}\right) = \log\left(\frac{N - df_t}{df_t}\right) \approx \log\left(\frac{N}{df_t}\right) \tag{10.12}$$

- theoretical justification for the **inverse document frequency** idf_t
- estimate cannot be computed as easily for p_t

Estimating p_t

- relevance feedback: use the frequency of term occurrence in known relevant documents
- ad-hoc retrieval:
  - assume p_t is constant over all terms x_t in the query and that $p_t = 0.5$

  $\Rightarrow$ p_t and $1 - p_t$ cancel out in the RSV
  - very weak estimate, but does not disagree with expectation that query terms appear in many but not all relevant documents
  - combined with approximation of u_t, the ranking is determined simply by which query terms occur in documents scaled by their idf weighting
  - yields okay results for short documents

10.6.6 Problems

- BMI does not pay attention to term frequency and document length

$\Rightarrow$ BMI is less robust on longer texts

10.7 Okapi Best Match 25 (BM25)

10.7.1 Idea

- incorporate term frequency and document length into BMI model

10.7.2 Basic Weighting

- Improve idf term [log N/df] by factoring in term frequency and document length.

$$RSV_d = \sum_{t \in q} \log \left[\frac{N}{\mathrm{df}_t} \right] \cdot \frac{(k_1 + 1)\mathrm{tf}_{td}}{k_1((1-b) + b \times (L_d/L_{\mathrm{ave}})) + \mathrm{tf}_{td}}$$

- tf_{td}: term frequency in document d
- L_d (L_{ave}): length of document d (average document length in the whole collection)
- k_1: tuning parameter controlling the document term frequency scaling
- b: tuning parameter controlling the scaling by document length
- Interpret BM25 weighting formula for $k_1 = 0$
- Interpret BM25 weighting formula for $k_1 = 1$ and $b = 0$
- Interpret BM25 weighting formula for $k_1 \mapsto \infty$ and $b = 0$
- Interpret BM25 weighting formula for $k_1 \mapsto \infty$ and $b = 1$

10.7.3 Improved Weighting

- For long queries, use similar weighting for query terms

$$RSV_d = \sum_{t \in q} \left[\log \frac{N}{\mathrm{df}_t} \right] \cdot \frac{(k_1 + 1)\mathrm{tf}_{td}}{k_1((1-b) + b \times (L_d/L_{\mathrm{ave}})) + \mathrm{tf}_{td}} \cdot \frac{(k_3 + 1)\mathrm{tf}_{tq}}{k_3 + \mathrm{tf}_{tq}}$$

- tf_{tq}: term frequency in the query q
- k_3: tuning parameter controlling term frequency scaling of the query
- No length normalization of queries (because retrieval is being done with respect to a single fixed query)
- The above tuning parameters should ideally be set to optimize performance on a development test collection. In the absence of such optimization, experiments have shown reasonable values are to set k_1 and k_3 to a value between 1.2 and 2 and $b = 0.75$

CHAPTER 11

Text Classification: Naive Bayes

11.1 How Search Engines Use Classification

- language identification (English vs German)
- automatic detection of spam documents
- sentiment detection (is a movie review positive or negative)
- topic-specific search (restrict search to documents about health)
- standing queries (Google Alerts)

11.2 Classification Methods

11.2.1 Manual Classification

- very accurate if job is done by experts
- consistent when problem size and team is small
- scaling is problematic and expensive

$\Rightarrow$ usually not suited for today's data problems

11.2.2 Rule Based Classification

- complex rules determine the class of a given document
- very high accuracy if rules are refined carefully over time by an expert
- building and maintaining rather complicated and expensive
- (e.g. used by Google Alerts)

11.2.3 Statistical/Probabilistic Classification

- see classification as a machine learning problem
- use supervised learning to learn a classifier
- no free lunch: requires hand-classified training data
- BUT: this manual classification can be done by none-experts
- e.g.: Naive Bayes

11.3 Formal Definition

- given:
  - **document space** $\mathcal{X}$: vector space which is used to represent documents
  - fixed **set of classes** $\mathcal{C} = \{c_1, \ldots, c_j\}$
  - a **training dataset** $\mathcal{D} \subseteq \mathcal{X} \times \mathcal{C}$ consisting of entries (d, c), representing a document labeled with its class
- idea:
  - use learning algorithm to learn a model $h : \mathcal{X} \to \mathcal{C}$ predicting the correct class for given document (representation)

11.4 The Naive Bayes Classifier

11.4.1 Concept

- compute probability of document d being in class c as

$$\mathbb{P}(c|d) \propto \mathbb{P}(c) \cdot \prod_{1 \le k \le n_d} \mathbb{P}(t_k|c) \tag{11.1}$$

  where

  - n_d: number of tokens in the document
  - $\mathbb{P}(t_k|c)$: conditional probability of term t_k occurring in a document of class c
    - measure of how much **evidence** t_k contributes that c is the correct class
    - if evidence is identical for all terms, $P(c)$ becomes the only important variable
  - $\mathbb{P}(c)$: prior probability of class c
- return best class according to **maximum a posteriori (MAP)** principle:

$$\arg\max_{c \in \mathcal{C}} \widehat{\mathbb{P}}(c|d) = \arg\max_{c \in \mathcal{C}} \widehat{\mathbb{P}}(c) \cdot \prod_{1 \le k \le n_d} \widehat{\mathbb{P}}(t_k|c) \tag{11.2}$$

  where all quantities with a hat are **estimates**

11.4.2 Practical Considerations

- computing the product can be arithmetically problematic (underflow when multiplying small probabilities)
- since log is a monotonic function, we can also take the log of the product without changing the ranking

⇒ in practice we compute

$$\arg\max_{c \in \mathcal{C}} \log \left(\widehat{\mathbb{P}}(c) \cdot \prod_{1 \leq k \leq n_d} \widehat{\mathbb{P}}(t_k|c) \right) = \arg\max_{c \in \mathcal{C}} \log \left(\widehat{\mathbb{P}}(c) \right) + \sum_{1 \leq k \leq n_d} \log \left(\widehat{\mathbb{P}}(t_k|c) \right) \tag{11.3}$$

- interpretation
  - each $\log\left(\widehat{\mathbb{P}}(c)\right)$ is a weight indicating the relative frequency of class c
  - complete term is measure of how much evidence there is for the document being in the class

11.4.3 Parameter Estimation

- prior:

$$\widehat{\mathbb{P}}(c) = \frac{N_c}{N} \tag{11.4}$$

  where

  - N_c: number of docs in class c
  - N: total number of docs
- conditional probabilities:

$$\widehat{\mathbb{P}}(t|c) = \frac{T_{c,t}}{\sum\limits_{t' \in V} T_{c,t'}} \tag{11.5}$$

  where

  - $T_{c,t}$: number of tokens of t in training docs of class c (counting multiple occurrences!)
- we use a **Naive Bayes assumption** here: we assume $\widehat{\mathbb{P}}(t|c)$ is independent of the terms position in the document!

Problems with 0s

- when one term of the product / log sum is 0, the whole probability gets 0 / is undefined!
  - happens easily if a vocabulary term t does not occur in any document of class c

  ⇒ we would never assign a document containing term t to c
- solution: **smoothing**

- simplest form: **add-1-smoothing**

$$\widehat{\mathbb{P}}(t|c) = \frac{T_{c,t}+\mathbf{1}}{\sum_{t'\in V}(T_{c,t'}+\mathbf{1})} = \frac{T_{c,t}+\mathbf{1}}{\left(\sum_{t'\in V} T_{c,t'}\right)+|V|} \tag{11.6}$$

11.4.4 Time Complexity

mode	time complexity
training	$\Theta(\lvert\mathbb{D}\rvert L_{\mathsf{ave}} + \lvert\mathbb{C}\rvert\lvert V\rvert)$
testing	$\Theta(L_{\mathsf{a}} + \lvert\mathbb{C}\rvert M_{\mathsf{a}}) = \Theta(\lvert\mathbb{C}\rvert M_{\mathsf{a}})$

- L_{ave}: average length of a training doc, L_{a}: length of the test doc, M_{a}: number of distinct terms in the test doc, $\mathbb{D}$: training set, V: vocabulary, $\mathbb{C}$: set of classes
- $\Theta(|\mathbb{D}|L_{\mathsf{ave}})$ is the time it takes to compute all counts.
- $\Theta(|\mathbb{C}||V|)$ is the time it takes to compute the parameters from the counts.
- Generally: $|\mathbb{C}||V| < |\mathbb{D}|L_{\mathsf{ave}}$
- Test time is also linear (in the length of the test document).
- Thus: Naive Bayes is linear in the size of the training set (training) and the test document (testing). This is optimal.

11.5 Derivation of Naive Bayes

11.5.1 Basics

- find the class that is most likely given the document

$$c^* = \arg\max_{c\in\mathcal{C}} \mathbb{P}(c|d) \overset{\text{Bayes}}{=} \arg\max_{c\in\mathcal{C}} \frac{\mathbb{P}(d|c)\cdot\mathbb{P}(c)}{\mathbb{P}(d)} \tag{11.7}$$

- since $\mathbb{P}(d)$ is constant for all classes it can be dropped:

$$c^* = \arg\max_{c\in\mathcal{C}} \mathbb{P}(d|c)\cdot\mathbb{P}(c) = \arg\max_{c\in\mathcal{C}} \mathbb{P}(\langle t_1,\dots,t_k\dots t_{n_d}\rangle|c)\cdot\mathbb{P}(c) \tag{11.8}$$

- extreme amount of parameters $\mathbb{P}(\langle t_1,\dots,t_k\dots t_{n_d}\rangle|c)$ (one for each combination of a class and a sequence of words)
  - to estimate this, we would need tons of data, which we usually do not have ⇒ problem of **data sparseness**
- **Naive Bayes conditional independence assumption** to reduce number of parameters to manageable size:

$$\mathbb{P}(d|c) = \mathbb{P}(\langle t_1,\dots,t_k\dots t_{n_d}\rangle|c) = \prod_{1\le k\le n_d} \mathbb{P}(X_k = t_k|c) \tag{11.9}$$

where

  - X_k: k'th term of the document

- further simplifying assumption for practical estimation: **positional independence**
  - probability of a term is identical for all positions: $\widehat{\mathbb{P}}(X_{k_1} = t|c) = \widehat{\mathbb{P}}(X_{k_2} = t|c)$
  - note that this assumption is made only for computing the practical estimates!

11.5.2 Violation of Assumptions

- conditional independence is usually **badly violated**: e.g. probability of "Bieber" coming after "Justin" a lot higher than after "Angela"
- positional independence is usually **badly violated** as well: e.g. "Hello" appears a lot more likely at the start of a sentence / doc
- NB is horrible at correctly estimating probabilities
- BUT: classification is about predicting the correct class and NOT correctly estimating probabilities

$\Rightarrow$ NB's estimates are good enough to get good class predictions

11.5.3 Positive Aspects

- robust to nonrelevant features (compared to some more complicated learning methods)
- robust to concept drift (changing of definition of class over time)
- better than methods like trees when there exist many equally important features
- good baseline for text classification (not the best, e.g. SVM is usually better)
- **optimal** if independence assumptions hold (true for some domains)
- very fast
- low storage requirements

11.5.4 View: Generative Model

- NB can be seen as a generative model
- we generate a class c with a probability of $\mathbb{P}(c)$
- generate each word, conditional on the class, but independent of each other with probability $\mathbb{P}(t_k|c)$
- to classify docs, we re-engineer this process and find the class that is most likely to have generated the doc

11.6 Evaluating Classification

- evaluation using **precision**, **recall** and **F-Measure**

	in the class	not in the class
predicted to be in the class	true positives (TP)	false positives (FP)
predicted to not be in the class	false negatives (FN)	true negatives (TN)

- confusion matrix gives only value for one class, but we want aggregate value over all classes
- Micro- vs. Macro Averaging Measure (F_1 measure)
  - **Macro Averaging**
    - compute F_1 for each class
    - compute average of these F_1 values
  - **Micro Averaging**
    - compute TP, FP, FN for each class
    - create sums of TP, FP, FN over all classes
    - compute F_1 for aggregate values
  - note: micro and macro-averaging are identical for precision and recall

CHAPTER 12

Link Analysis

12.1 Anchor Texts

12.1.1 Web as Directed Graph

- model web as directed graph where links are edges and pages are nodes
- assumptions we make
    1) hyperlink is a quality signal
        - not true in general (e.g. author might be gossiping, spam links etc.)
    2) given a link $d_1 \rightarrow d_2$: anchor text describes content of d_2
        - anchor text is used loosely for the text surrounding the hyperlink (more than the text in the "a"-tag)
        - also no true in general (e.g. authors might use horrible descriptions for links)

12.1.2 Use of Anchor Texts

- searching on [text of d_2] + [anchor text $\rightarrow d_2$] is often more efficient than searching on [text of d_2] only
    - e.g. query: "IBM"
    - matches copyright page, many spam pages, Wikipedia article, but maybe NOT main page (consisting mainly of images!)
    - using the anchor text additionally, the main page might be found a lot easier
- anchor text is often a better description of a pages content than the page itself
- anchor texts can be weighted more highly than document text (based on the assumptions made earlier)

12.1.3 Origins of PageRank: Citation Analysis

- citation frequency can be used to measure impact of a scientific article
- usually one uses **weighted** citation frequency
  - citation's vote is weighted according to its citation impact
- appropriately weighted citation frequency is an excellent measure of quality both for web pages and scientific publications

12.2 PageRank

12.2.1 Underlying Model: Random Walk

- assume web surfer doing a random walk on the web (start at random and move along links at random)
- at each step, the surfer is at exactly one page
- when running this long enough, each page has a **long-term visit rate**, called the **PageRank**
- **long-term visit rate** of page d: probability that a web surfer is at page d at a given point in time

Formalization: Markov Chain

- consists of
  - N states
  - an $N \times N$ transition probability matrix P
- $P_{i,j}$:= probability of moving to state j given we are in state j
- for all i: $\sum_{j=1}^{N} P_{i,j} = 1$
- state := page

Ergodic Markov Chains

- for computing the steady-state probability (:= long term visit rate), chain has to be **ergodic**
  - **ergodic** ⇔ **irreducible** and **aperiodic**
  - **irreducible**: there is a path from any page to any other
  - **aperiodic**: pages cannot be partitioned s.t. the random walker visits the partitions sequentially
- steady-state probability (ssp): over a long period of time, we visit each state with its ssp

12.2.2 Making the Web-Graph Ergodic

- direct Markov Chain of web graph is non-ergodic due to **dead ends** (page without any outgoing link)

⇒ irreducibility is violated

- solution: **teleportation**
  - at a dead end: jump to a random webpage with probability $1/N$ (N: number of web pages)
  - at a non-dead end:
    - with probability $(1-\epsilon)$: follow one of the outgoing links at random
    - with probability ϵ: jump to any webpage at random
  - ϵ: **teleportation rate**
- teleporting makes the graph ergodic!

12.2.3 Visiting as Probability Vector

- formalization of visit process
- probability vector: $x_t = (x_{t,1}, \dots, x_{t,N})$ (N: number of states)
  - interpretation: random walk is in step t at state i with probability x_i
  - initial value: usually initial distribution or uniform distribution across all states
- update formula to get step $t+1$:

$$x_{t+1} = x_t \cdot P \tag{12.1}$$

12.2.4 Computation of PageRank: Underlying Problem

- $\pi = (\pi_1, \dots, \pi_N) :=$ page rank vector of steady-state probabilities
- solving the following equation gives the page rank vector:

$$\pi = \pi \cdot P \tag{12.2}$$

- π is the **principal left eigenvector** for P (left eigenvector with the largest eigenvalue)
- transition probability matrices have an eigenvalue of 1

12.2.5 Computation of PageRank: Power Method

1) start with any distribution vector x_0 (e.g. uniform distribution over states)
2) compute

$$x_{t+1} = x_t \cdot P \tag{12.3}$$

   until $x_{t+1} = x_t$
3) after enough iterations we reach the steady state (asymptotically)

12.2.6 Query Processing using PageRank

- retrieve pages satisfying query
- rank them by their page rank (in decreasing order as: higher page rank = better)
- return ranked results

12.2.7 Issues

1) real surfers are not random surfers
   - Markov model is not a good model for surfing, but good enough
2) simple PageRank ranking produces bad results for many pages
   - e.g.: query "video service"
   - Yahoo page has a high PageRank and contains the terms $\Rightarrow$ top ranked result (although it offers no video service)

   $\Rightarrow$ in practice: rank according to weighted combination of raw and anchor text match, PageRank and other factors

12.2.8 Importance of PageRank

- not the only important factor anymore
- rumor: negligible effect on ranking by now, but still essential part

12.3 HITS: Hyperlink-Induced Topic Search

12.3.1 Idea

- two types of relevance (i.e. pages) on the web
  1) **hubs**: hub page is good list of links to pages answering the information need
     - good hub page (for a topic) **links to** by many authorities (of the topic)
  2) **authorities**: authority page is a direct answer to the information need
     - good authority page (for a topic) **is linked to** by many hubs (of the topic)
- **aim**: given a query, compute one ranked list of authorities and one of hubs
  - done by computing **authority score** $a(d)$ and **hub score** $h(d)$ for all web pages d and sorting them by these scores

12.3.2 Algorithm

assume we are given a query

1) **root set**:= result set of a regular web search
2) **base set** B:= **root set** + all pages which are linked to or link to pages in the root set
3) for all pages d in the base set
   a) $h(d) = 1$ and $a(d) = 1$
   b) until convergence compute for all $d \in B$

$$\begin{aligned} h(d) &= \sum_{y \in B: d \rightarrow y} a(y) \\ a(d) &= \sum_{y \in B: y \rightarrow d} h(y) \end{aligned} \tag{12.4}$$

4) output pages (or subset) of base set ranked by hub scores
5) output pages (or subset) of base set ranked by authority scores

12.3.3 Remarks

- a and h values can become really big

$\Rightarrow$ scale them down after each iteration by some factor
  - okay to do, as we are only interested in their ranking
- algorithm usually converges after few iterations
- HITS can pull together good pages regardless of content
- once we have a base set, it only performs **link analysis**
- danger: **topic drift**
  - pages found by the links in the root set may not be related to the original query

12.3.4 HITS as Eigenvector Problem

- define adjacency matrix $A = N \times N$ (N: number of pages in base set)

$$A_{i,j} = \begin{cases} 1 & \text{if page } i \text{ links to page } j \\ 0 & \text{else} \end{cases} \tag{12.5}$$

- define $h = (h_1, \ldots, h_N)$ / $a = (a_1, \ldots, a_N)$ as the hub / authority value vector, storing the according values for all pages
- HITS update formula

$$\begin{aligned} h &= A \cdot a \\ a &= A^T \cdot h \end{aligned} \tag{12.6}$$

- substitution yields

$$\begin{aligned} h &= A \cdot A^T \cdot h \\ a &= A^T \cdot A \cdot a \end{aligned} \tag{12.7}$$

⇒ final h is **eigenvector** of $A \cdot A^T$

⇒ final a is **eigenvector** of $A^T \cdot A$

⇒ **HITS also uses the power method**

12.4 PageRank vs HITS

- both algorithms can be formulated as eigenvector problems
- PageRank can be precomputed, whereas HITS as to be computed at query time
  - HITS is too expensive in most applications
- different design choices concerning
  - eigenvector problem formalization
  - set of pages to apply the approach to
- claim: on the web, a good hub is usually also a good authority

⇒ difference between PageRank and HITS ranking usually not too different